Design Space Exploration and Optimization of Embedded Memory Systems

A Thesis
Presented to
The Academic Faculty

by

Rodric Michel Rabbah

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
August 2006

Design Space Exploration and Optimization of Embedded Memory Systems

Approved by:

Dr. Krisha Palem, Advisor
School of Electrical and
Computer Engineering
Georgia Institute of Technology

Dr. Roy Dz-Ching Ju
Advanced Micro Devices

Dr. Santosh Pande
College of Computing
Georgia Institute of Technology

Dr. Yannis Smaragdakis
College of Computing
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and
Computer Engineering
Georgia Institute of Technology

Date Approved: May 17, 2006

to my family

ACKNOWLEDGEMENTS

This thesis would not exist without the help and advice of many people. I am grateful for my advisor, Krishna Palem, for giving me the freedom and resources to pursue research that interested me, and for his unceasing mentorship and guidance throughout my graduate life.

I am also indebted to those who donated their time to comment on this work, and particularly the members of my thesis committee, Roy Ju, Santosh Pande, Yannis Smaragdakis, and Sudhakar Yalamanchili.

Many thanks to the members of the ReaCT-ILP and CREST groups. Benjamin Goldberg, Hansoo Kim, Allen Leung, Amit Nene, Igor Petchanski, and Suren Talla, were incredible mentors at NYU. Their support and nurturing made my early research exciting. The CREST gang made Atlanta bearable with great debates and great company. I am especially lucky to have worked with Lakshmi Chakrapani, Mongkol Ekpanyapong, Charles Hardnett, Jinwoo Kim, Pinar Korkmaz, Tushar Kumar, and Kiran Puttaswamy. Lakshmi is a close friend with whom I enjoyed daily interactions and the exchange of ideas. Jinwoo tolerated my impeccable tardiness everyday as we commuted to school. Both are great friends and colleagues. I was also fortunate to have worked with Weng-Fai Wong during his sabbatical year at CREST. He was, and remains, as much an advisor as a colleague.

I am also thankful for the internship opportunities that Bob Rau and Josh Fisher offered me during my graduate career. My internships were all the more rewarding because of their incredible insights and visions. I am also lucky to have worked closely with the members of their respective groups, especially Giuseppe Desoli, Paolo Faraboschi, Scott Mahlke, Mike Schlansker, and Rob Schreiber.

Finally, I thank my parents for having prepared me for my endeavors. I owe them all my success.

This research was funded in part by DARPA contract F33615-99-1499, and awards from Hewlett-Packard Laboratories and Yamacraw.

TABLE OF CONTENTS

DEDICATION . . . . . **iii**

ACKNOWLEDGEMENTS . . . . . **iv**

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **viii**

SUMMARY . . . . . **ix**

I INTRODUCTION **1**

1.1 Methodology and contributions 4

1.2 Thesis organization 5

II FOUNDATION AND ANALYTICAL MODEL **7**

2.1 Model of demand bandwidth 8

2.2 Implications to cache design 9

2.3 Temporal locality 11

2.4 Spatial locality 12

2.5 Locality enhancing optimizations 14

2.6 Implication to memory system design 15

2.7 Other considerations 16

2.8 Directions for future work 18

2.9 Summary and remarks 19

III DATA REMAPPING **20**

3.1 Remapping with offset manipulation 22

3.2 Pointers and alias analysis 25

3.3 Remapping of array objects 26

3.4 Profile analysis 27

3.5 Limitations 28

IV DESIGN SPACE EXPLORATION **29**

4.1 Performance evaluation 30

4.2 Design space optimization 32

V INTRODUCTIVE PREFETCHING 35
5.1 Introspective prefetching example 37
5.2 Implementation details 40
5.3 Introspective execution 42
5.4 Architecture considerations 44
5.5 Memory system performance 44
5.5.1 Methodology 45
5.5.2 Evaluation 47
5.5.3 Precomputation overhead 50
5.6 Summary and concluding remarks 50
VI CONCLUDING REMARKS 51
REFERENCES 53

LIST OF TABLES

1 Summary of design trade-offs. 10
2 Locality and bandwidth interactions. 11
3 Benchmarks, workloads and main memory footprints. 30
4 Data remapping speedup summary. 30
5 Results for Pentium 3 processor. 31
6 Results for Pentium 2 processor. 31
7 Results for UltraSparc II processor. 31
8 Benchmarks for design space exploration. 32
9 Execution cycles and energy results before remapping. 33
10 Execution cycles and energy results after remapping (relative to Table 9). . 33
11 Execution cycles and energy results after remapping (relative to Table 9). . 33
12 Execution cycles and energy results after remapping (relative to Table 9). . 33
13 Delinquent loads in different benchmarks. 36
14 Benchmarks and input workloads. 45
15 Simulated processor. 46

LIST OF FIGURES

1 Design space exploration and compiler optimizations as duals. 3
2 Summary of analytical framework. 5
3 Performance versus bandwidth. 8
4 Examples of spatial locality. 13
5 Example locality enhancing optimization for multi-dimensional arrays. . . . 15
6 Example locality enhancing optimization for heap objects. 15
7 Example prefetching process. 17
8 Example control flow trace. 19
9 Physical working set size after remapping. 23
10 Pool allocation. 23
11 An example pool allocator. 24
12 Layout according to different offset calculations. 24
13 Runtime disambiguation of offset expressions. 25
14 Algorithm to compute neighbor affinity. 27
15 Example pointer-chasing code. 37
16 Example precomputation and prefetching. 38
17 Example LDC unrolling. 39
18 Time line for a precomputation distance of three iterations. 39
19 Introspective execution. 40
20 Percent reduction in memory stalls. 47
21 Example LDC prefetching without introspective instructions. 48
22 Speedup due to introspective instructions. 49

SUMMARY

Recent years have witnessed the emergence of microprocessors that are embedded within a plethora of devices used in everyday life. Embedded architectures are customized through a meticulous and time consuming design process to satisfy stringent constraints with respect to performance, area, power, and cost. In embedded systems, the cost of the memory hierarchy limits its ability to play as central a role. This is due to stringent constraints that fundamentally limit the physical size and complexity of the memory system. Ultimately, application developers and system engineers are charged with the heavy burden of reducing the memory requirements of an application.

This thesis offers the intriguing possibility that compilers can play a significant role in the automatic design space exploration and optimization of embedded memory systems. This insight is founded upon a new analytical model and novel compiler optimizations that are specifically designed to increase the synergy between the processor and the memory system. The analytical models serve to characterize intrinsic program properties, quantify the impact of compiler optimizations on the memory systems, and provide deep insight into the trade-offs that affect memory system design.

CHAPTER I

INTRODUCTION

During the past three decades, the microprocessor has proliferated many aspects of daily life with a scope and depth that was hard to imagine during its early development. Today, processors are found in mainframes, servers, desktop workstations, and a plethora of embedded products that include sensors, robots, gaming consoles, media players, digital cameras, network routers, navigation systems, mobile phones and other portable devices.

For microprocessors, the periodic doubling in the number of transistors that can be fabricated on a chip often meant a hundred percent increase in performance every year and a half, at no additional cost. The phenomenon, first prophesied by Intel co-founder Gordon E. Moore, has *"delighted consumers and product designers, and has been the main reason why the microprocessor has been one of the greatest technologies of our time"* [8]. Every generation of processors has enabled new, larger, and more complex applications.

The leading microprocessor companies have focused on maintaining the Moore trajectory even though the National Technology Roadmap for Semiconductors has observed that the number of transistors that engineers could design into new circuits is increasing at a rate of only twenty percent a year [25]. This low rate is due to the physical realities of wire delay and power consumption that challenge all existing design methodologies. As a result, the industry is beginning to witness a significant paradigm shift in processor design, away from large monolithic chips, toward multicore processors and heterogeneous systems on a chip.

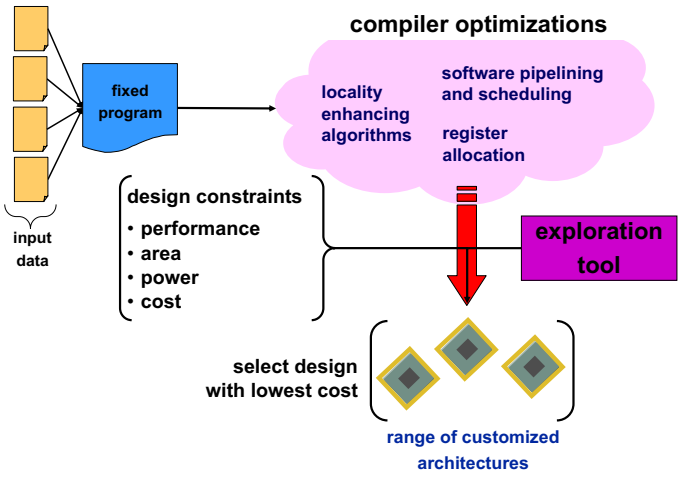
The departure from monolithic processors is emphasized in the embedded domains where the increasing transistor density is used to provide a rich set of highly specialized devices, each customized for a specific computational task. Hence, while a specialized core is used to carry out number crunching functions, a general purpose processor is used for less computationally demanding tasks. This methodology leverages the latest commercial-off-the-shelf technology in order to successfully design custom solutions within the short time-to-market

window characteristic of the industry. This is in contrast to the traditional approach to customization which required significant time and financial investments to design application specific integrated circuits (ASICs). The ASIC methodology incurs high non-recurring engineering costs, and is only practical in high-volume applications where the costs can be effectively amortized.

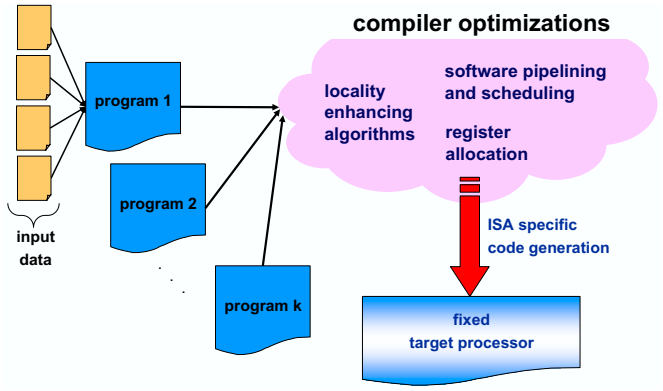
Custom processing solutions provide substantial opportunities for performance gains. They also magnify the need for sustainable high rates of data delivery from main memory to the processing units. The memory hierarchy has served as a central component in computing platforms since the introduction of the von Neumann machine [9, 51]. It uses several levels of cache memories, with each level trading off capacity for access speed, to bridge the widening performance gap between processor and main memory: while processing speeds have nearly doubled every year, memory response times have increased a much slower rate of 7% a year [22].

In embedded systems, the cost of the memory hierarchy limits its ability to play as central a role. This is due to stringent area, power, and cost constraints that fundamentally impact design choices, and limit the physical size and complexity of the memory system. Ultimately, application developers and system engineers are charged with the burden of reducing the memory requirements of an application in order to avoid memory bottlenecks that degrade processor throughput.

This thesis presents a foundation for the automatic optimization of memory requirements using novel compiler techniques that are designed to increase the synergy between the processor and its memory system. The thesis offers the intriguing possibility of compiler optimizations playing a significant role in optimizing the memory design of an embedded system. As shown in Figure 1(a), design space exploration involves exploring alternate architectural solutions to meet a specified performance constraint for a fixed program $\mathcal{P}$. Thus while the program is fixed, the optimization techniques are applied to find the architecture that best satisfies the desired constraints. In contrast and as shown in Figure 1(b), the dual of this problem is the domain of a traditional compiler optimization, wherein the



(a) Design space exploration.



(b) Traditional compiler optimization.

Figure 1: Design space exploration and compiler optimizations as duals.

applications or programs $\mathcal{P}_1, \mathcal{P}_2 \ldots \mathcal{P}_k$ are optimized to achieve the best possible performance on a fixed target processor. An exploratory design space optimization demonstrates a compiler-guided two fold reduction in the size of a memory hierarchy.

There are mainly two classes of optimizations that can improve the memory system performance. The first is a class locality enhancing optimizations that improve temporal and spatial clustering of addresses to leverage the design characteristics of modern memory systems and reduce memory traffic. These optimizations are founded upon fundamental intrinsic properties of all programming paradigms. The second is a class of latency hiding

optimizations that anticipate future memory references and prefetch them into the fastest memories ahead of their actual use. Such optimizations have long served to improve performance by mitigating the gap between processor and memory speeds.

This thesis introduces data remapping as a new locality enhancing optimization, and introspective prefetching as a new latency hiding optimization. Both advocate a model of introspective computation where a compiler orchestrates macroscopic execution policies, and the processor dynamically adapts the policies locally (at microscopic scale) in response to runtime events that cannot be practically anticipated and resolved at compile time. This results in a more effective use of the memory hierarchy, along with smaller memory footprints and shorter access latencies. These improvements not only directly translate to faster execution, but more importantly, they also impact the physical characteristics of the memory hierarchy by reducing its size, power needs, and overall cost.

1.1 Methodology and contributions

At the heart of this work is a compile time data remapping algorithm that is designed to enhance locality for dynamic programs with irregular memory access patterns. Stated in simple terms, remapping is a reorganization of the application data, such that elements that are accessed contemporaneously are in fact placed together in memory. In other words, data remapping aims to improve the spatial locality of objects that have strong temporal affinities for each other. It leads to significant performance gains on simulated and existing commodity processors. It is also shown to reduce the memory system requirements in half for some benchmarks, leading to significantly less costly memory hierarchies.

Another contribution of this work is the concept of introspective prefetching. It is a unified compiler-orchestrated strategy for effective latency hiding techniques. Introspective prefetching is an innovative combination of speculative and predicated execution which are traditionally used to increase instruction level parallelism. In this thesis, they are used to dynamically modify execution in response to information propagated from the memory system. This contribution facilitates the introduction of effective prefetching strategies into embedded memory systems. Introspective prefetching is shown to be quite effective for

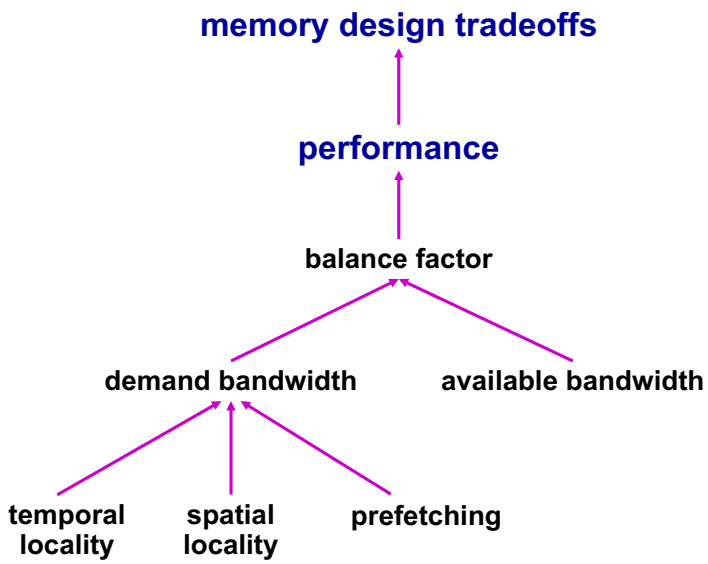


Figure 2: Summary of analytical framework.

various types of applications. It has negligible architectural overhead, and is far less costly that existing latency hiding strategies.

This thesis introduces a new analytical foundation to formally reason about memory design space trade-offs, and to facilitate an exploration framework centered around compiler technology. The analytical framework is summarized in Figure 2. It characterizes the intrinsic behavior of a program and can interpret those properties in a microarchitecture context. The models quantify well known concepts of temporal and spatial locality. The same models are shown for the first time to not only explain performance implications, but also serve to analyze the impact of locality on memory system design. Previous work on locality models is focused on one dimension or the other. The analytical models can measure the impact of compiler optimizations on memory system performance, and provide deep insight into the trade-offs that affect memory system design.

1.2 Thesis organization

The inspiration for this thesis is a desire for an analytical framework that serves to quantify the design trade-offs that affect memory systems. Chapter 2 introduces the analytical foundation for this thesis.

Chapter 3 introduces data remapping, a new locality enhancing algorithm design to improve the spatial locality in applications that are rich in dynamic data structures.

Chapter 4 presents an evaluation of data remapping using commodity processors, and demonstrates that the performance gains that data remapping affords can be traded to optimize the memory architecture.

Chapter 5 introduces and evaluates the concept of introspective prefetching, a low cost latency hiding strategy that is practical for embedded processors.

Finally, Chapter 6 identifies areas for future work and concludes the thesis.

CHAPTER II

FOUNDATION AND ANALYTICAL MODEL

There are many optimizations that impact the design criteria of a memory system. This chapter presents an analytical model that serves as a foundation for characterizing the impact of memory optimizations on the design space. The model serves as a guide to understanding design trade-offs, and facilitates an exploration framework centered around compiler technology.

The model is based on the insight that the realized properties of an application (e.g., performance in total cycles) are a manifestation of intrinsic program characteristics in the context of a specific architecture. The model decouples the intrinsic and realized behaviors, and demonstrates how to independently quantify the intrinsic properties and interpret them in a physical setting. An important implication of the model is that memory optimizations that impact performance directly translate to optimizations of the memory design space in terms of its size, power, and cost.

The model is built on the premise that bandwidth is the main bottleneck to faster performance. For the purpose of this exposition, bandwidth is simply the rate of data transfer between two components of the memory systems (e.g., main memory and cache). As illustrated in Figure 3, when the available bandwidth is saturated, any further increase in the clock frequency does not improve the processing rate [18]. Compiler optimizations that target the memory system directly impact the bandwidth requirements of the application. The analytical model demonstrates how to quantify the demand bandwidth of the application, and can explain why memory optimizations are effective.

The demand bandwidth is intrinsic to the application, whereas the available bandwidth is a fixed architecture quantity. It is evident that the balance between demand and available bandwidth dictates the processing rates of the application. Performance degrades when the demand bandwidth exceeds the available bandwidth because the processor spends more

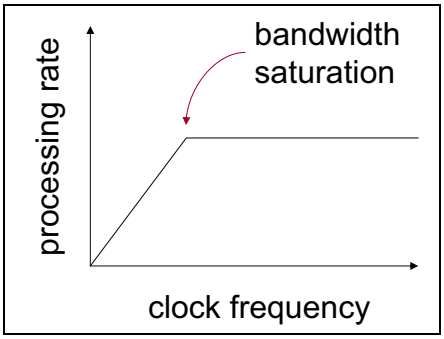


Figure 3: Performance versus bandwidth.

time waiting for data delivery. In contrast, the memory system ceases to be a bottleneck when the available bandwidth is high enough to satisfy the demand bandwidth.

2.1 Model of demand bandwidth

The demand bandwidth of an application is dependent on the application working set, or in other words, the set of addresses that are referenced during the program lifetime. The application working set is often much larger than any feasible cache or local memory[1] can accommodate, and hence it is necessary to consider intermediate working sets that span different intervals of execution. Thus, consider a reference trace T for a given application. The trace is the sequence of addresses referenced by the application during a specific execution. Now partition T into smaller non-overlapping traces s_1, s_2, ..., s_n, such that $\mathrm{T} = s_1|s_2|\ldots|s_n$.

The working set at the ith interval of execution is defined as $\hat{s}_i$. It is the set formed from the unique addresses in s_i. Let $\mathrm{L}(i)$ equal the number of references in s_i (i.e., $\mathrm{L}(i) = |s_i|$), and $\mathrm{V}(i)$ equal the number of unique addresses in that trace (i.e., $\mathrm{V}(i) = |\hat{s}_i|$). Naturally, the demand bandwidth for working set $\hat{s}_i$ is directly proportional the amount of data necessary to turnover (transform) working set $\hat{s}_{i-1}$ into $\hat{s}_i$. Formally, the turnover factor of the ith working set is defined as

$$\Gamma(i) = \mathrm{V}(i) - |\hat{s}_i \cap \hat{s}_{i-1}| \tag{1}$$

[1]For clarity, the remainder of the chapter will simply refer to caches although the same conclusions and remarks apply to local memories.

where $\hat{s}_0$ is the empty set. The turnover is in terms of an addressable unit (au) such as a byte. That is, $\hat{s}_i$ has $\Gamma(i)$ bytes that are new compared to the previous working set, and each new byte requires an expensive memory access.

A working set can amortize the cost of memory accesses over its execution interval. Here, the duration of a working set is measured with respect to the length of its corresponding trace. The demand bandwidth is hence defined as

$$\mathcal{D}(i) = \frac{\Gamma(i)}{\mathrm{L}(i)} \tag{2}$$

for any positive i less than or equal to n. A working set spanning a long period of time (i.e., $\mathrm{V}(i) \ll \mathrm{L}(i)$) requires less bandwidth than a working set that is changing very rapidly (i.e., $\mathrm{V}(i) \approx \mathrm{L}(i)$). Thus, $\mathcal{D}$ is a measure of the amortization rate. It is in terms of aus (numerator) per reference (denominator), and ranges from a minimum value of 0 to a maximum value of 1 au per reference.

2.2 Implications to cache design

Stored-program architectures store program instructions and data in large memory structures. Caches obviate the need for long memory accesses by locating frequently used instructions and data closer to the processing units. On each cache reference, the cache searches its contents for the requested address. This is typically achieved by partitioning the address into two (or more) fields. The first, traditionally consisting of the low-order bits, is known as the index. It is used to reference one the corresponding sets in the cache. The second field, known as the tag, is compared against the tag of the referenced cache set. When the two tags match, an event known as a cache hit, the data in the referenced cache set is forwarded to the processor. In the event of a cache miss, where the address tag and the set tag do not match, the cache forwards the address to the next cache in the hierarchy, ultimately reaching main memory. When the request is serviced, the cache updates its contents and overwrites the appropriate set to store the new data and tag field.

Caches have a finite capacity, and their efficacy is dependent on the intrinsic properties of the application working sets. For example, caches are ineffective when the demand bandwidth is near its maximum value. In this regime, nearly every reference in the working

Table 1: Summary of design trade-offs.

	Ratio of actual to intrinsic demand bandwidth	**Performance implications**
$C \approx \mathrm{V}(i)$	$= 1$	no slowdown
$C < \mathrm{V}(i)$	≥ 1	may slowdown
$C > \mathrm{V}(i)$	≤ 1	may speedup

set requires an independent access to (fetch from) memory. When this scenario is observed, it is prudent to bias the design choice toward higher bandwidth and smaller caches. When the demand bandwidth is low, the size of a working set directly impacts the cache size. For a fixed cache size C, there a three scenarios.

- If $C \approx \mathrm{V}(i)$ then the intermediate working set can completely reside in the cache. Any data elements that are unique to the current working set are fetched only once during its execution interval, and no additional memory accesses are necessary.

- If $C < \mathrm{V}(i)$ then the intermediate working set is larger than the cache capacity. This scenario increases the actual demand bandwidth as data elements from the current working set may be fetched on more than one occasion. The extent of the increase in bandwidth requirements is dependent on the nature of data reuse within the working set.

- If $C > \mathrm{V}(i)$ then the cache capacity is larger than the current working set, and as a result, it may have remnants of previous working sets. The data artifacts can serve to reduce the actual demand bandwidth because of greater reuse. To see that this is true, it is possible to redefine Equation 1 as

$$\Gamma(i, w) = \mathrm{V}(i) - |\hat{s}_i \cap (\hat{s}_{i-1} \cup \hat{s}_{i-2} \ldots \cup \hat{s}_{i-\omega})| \tag{3}$$

  where $\omega = min(w, i)$ is the length of the history to consider. When $w = 1$, Equations 1 and 3 are equivalent. At the extreme where $w = n$, the entire execution history is used.

Equation 3 shows that it is possible to reduce the turnover rate by preserving more and more history as the application progresses in its execution. Therefore it is natural to favor

Table 2: Locality and bandwidth interactions.

Temporal Locality	Turnover	Demand Bandwidth
high	low	low
low	high	high

larger caches during the design process. The objective of design space exploration is to find the smallest C such that the overall performance degrades no more than a specified threshold. For example, a design exploration process may search for the smallest C that is greater than or equal to 90% of the intermediate working set sizes. Alternatively, it may impose a partitioning of the application trace T into working sets of a fixed size $\mathcal{V}$ (i.e., $\mathrm{V}(i) = \mathcal{V}$ for all $1 \leq i \leq n$). In this case, the design exploration process is reduced to finding the smallest $\mathcal{V}$ that yields the lowest mean demand bandwidth.

Table 1 summarizes the design trade-offs. The model for demand bandwidth provides a foundation for quantifying the design choices, and pruning the design space systematically. The remainder of this chapter addresses other implications to memory system design.

2.3 Temporal locality

The turnover factor is related to the well known concept of temporal locality. A reference is said to have temporal locality if it occurs repeatedly within some time interval. Temporal locality reduces bandwidth requirements because of greater data reuse. This directly follows from Equation 1. If the temporal locality between two working sets is high, then their intersection is large and hence the packing factor is low. This ultimately contributes to reducing the demand bandwidth. It is thus no wonder that so many researchers have focused on computation reordering as a key locality enhancing optimizations (see [5] for a comprehensive survey). In such optimizations, a compiler attempts to reorder execution such that adjacent intervals of computation share common references.

The relationship between temporal locality and demand bandwidth is summarized in Table 2. Temporal locality of the ith working set is defined as

$$\mathrm{TL}(i) = 1 - \frac{\Gamma(i)}{\mathrm{L}(i)} \tag{4}$$

or simply one minus the number of turnover in references from one working set to another,

divided by the total number of references. When $\Gamma(i) \approx \mathrm{L}(i)$, the temporal locality approaches zero. This is as expected since the working set does not have any address reuse. In contrast, the locality is near perfect when $\mathrm{L}(i) \gg \mathrm{V}(i)$.

It may be tempting to distinguish between inter and intra working set locality. However note that when the inter locality is high, intra locality is inconsequential because many references in working set $\hat{s}_i$ are common to the previous working set $\hat{s}_{i-1}$ (i.e., $\Gamma(i) \approx 0$). This is true even when every reference in $\hat{s}_i$ is unique. When the inter locality is low, $\Gamma(i) \approx \mathrm{V}(i)$ and Equation 4 naturally quantifies intra working set locality. The demand bandwidth is formally related to temporal locality by the following equation

$$\mathcal{D}(i) = 1 - \mathrm{TL(i)} \tag{5}$$

showing that locality has a direct impact on the bandwidth requirements of an application.

2.4 Spatial locality

A concept related to temporal locality is that of spatial locality. A reference that is in close physical proximity (in the address space) to neighboring references (in the access trace) is said to have spatial locality. The concept of spatial locality is deeply rooted in many computation paradigms. As a simple example consider that instruction are fetched sequentially from memory even though they may execute out of order. As another example consider a loop that iterates through every element of a single dimensional array. The elements of the array are located in a contiguous region of memory. The working set of the loop is the array, and the references to the elements are all spatially clustered withing the same region of physical memory.

The notion of spatial locality is so well ingrained in computing paradigms that architectures are designed to transfer data in terms of blocks [18]. The block-fetch strategy serves to amortize the cost of an expensive memory access. It leverages the capabilities of modern memories to quickly and efficiently retrieve adjacent data items. Its efficacy however relies on the underlying premise that the reference that induced the transfer has adequate spatial locality. Otherwise the transfer leads to wasted bandwidth.

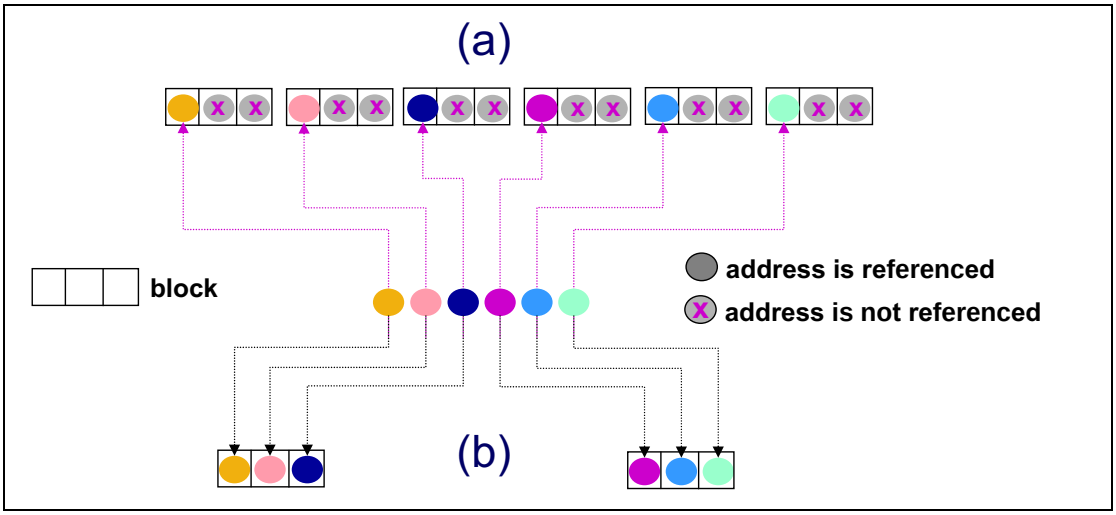


Figure 4: Examples of spatial locality.

Every address in the address space is a member of a specific physical block in memory. Usually, an address α belongs to block (α mod B) for a block of size B *au*s. An example is illustrated in Figure 4. The working set consists of six unique references, and a block consists of three addressable units. In (a), every address is shown to map to a different block, resulting in poor spatial locality. When these blocks are fetched, $\frac{2}{3}$ of their elements constitute pollution. In the alternate mapping shown in (b), the references have better spatial locality, mapping into two blocks. There is no pollution in this case, and the block fetch strategy is most effective.

The mapping of references into blocks introduces the notion of a physical working set size. The physical working set size is defined as

$$\mathrm{P}(\hat{s}, B) = \mathrm{R}(\hat{s}, B) \times B \tag{6}$$

where $\mathrm{R}(\hat{s}, B)$ equals the number of blocks required to map every address $\alpha \in \hat{s}$ to its corresponding block. The virtual working set size of $\hat{s}_i$ is defined as $\mathrm{P}(\hat{s}_i, 1)$ and it is equal to $\mathrm{V}(i)$ since $\mathrm{R}(\hat{s}_i, 1) = |\hat{s}_i|$. The turnover from one working set to another is now meaningfully expressed in terms of the physical working set as

$$\Gamma(i, B) = \mathrm{P}(\hat{s}_i, B) - \mathrm{P}(\hat{s}_i \cap \hat{s}_{i-1}, B) \tag{7}$$

which reduces to Equation 1 for $B = 1$ *au*.

The spatial locality of a working set $\hat{s}_i$ is defined as

$$\mathrm{SL}(i, B) = \frac{\mathrm{V}(i)}{\mathrm{R}(\hat{s}_i, B) \times B} \tag{8}$$

for any given working set. It is simply the ratio of virtual to physical working set sizes. Spatial locality impacts demand bandwidth as follows

$$\begin{aligned}
\mathcal{D}(i) &= \frac{\Gamma(i)}{\mathrm{L}(i)} \\
\mathcal{D}(i) &= \frac{\mathrm{P}(\hat{s}_i, B) - \mathrm{P}(\hat{s}_i \cap \hat{s}_{i-1}, B)}{\mathrm{L}(i)} \\
\mathcal{D}(i) &\leq \frac{\mathrm{P}(\hat{s}_i, B)}{\mathrm{L}(i)} \\
\textit{and since } \mathrm{P}(\hat{s}_i, B) &= \frac{V(i)}{\mathrm{SL}(i, B)} \\
\mathcal{D}(i) &\leq \frac{\mathrm{V}(i)}{\mathrm{SL}(i, B) \times \mathrm{L}(i)}
\end{aligned} \tag{9}$$

which implies that demand bandwidth is inversely proportional to spatial locality.

2.5 Locality enhancing optimizations

As is the case for temporal locality, researchers have focused a great deal of effort on optimizations that improve spatial locality. These optimizations alter the computation or the data layout such that data elements that are accessed contemporaneously are in fact placed together in memory. Two example optimizations are shown in Figures 5 and 6. They emphasize optimizations that improve the locality of data accesses. There are equivalent optimizations that improve the locality of instructions, but these are less relevant to this thesis.

The first example is a loop optimization for array data structures. Here, a two-dimensional array is organized in row-order, and a memory block contains elements `A[i,0]`, ..., `A[i,3]`. If the loops sweep through the data in column order, than the block fetch strategy is defeated. This leads to poor spatial locality, wasted bandwidth, and ultimately bad performance as the demand bandwidth models suggest. When the loops are interchanged, the inner loop sweeps through the data in row-order. The interchange improves spatial locality and leads to better performance because the block fetch strategy is working effectively.

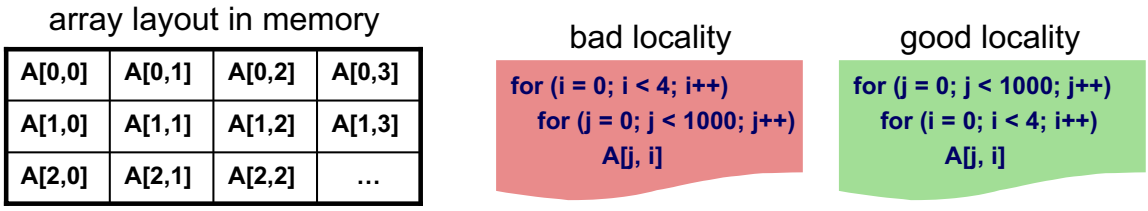


Figure 5: Example locality enhancing optimization for multi-dimensional arrays.

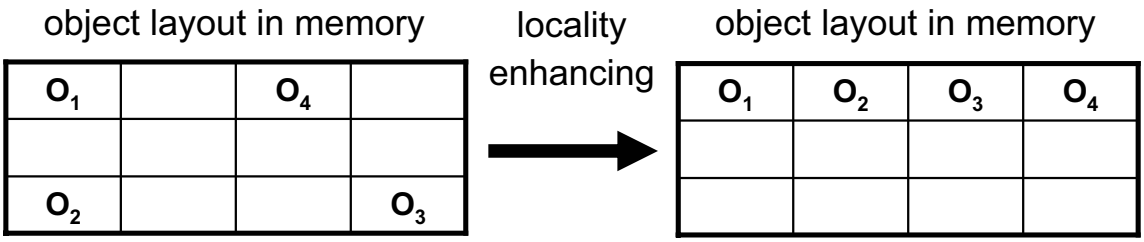


Figure 6: Example locality enhancing optimization for heap objects.

A second example highlights data reorganization for heap data structures. Heap objects are dynamically allocated, and their layout in memory may not suite the computation patterns of the application. A locality enhancing optimization can attempt to discover the object access patterns, and use the information to spatially cluster objects that have temporal affinity. The improved spatial locality ultimately leads to better performance.

The analytical models presented in this chapter can directly quantify the impact of locality enhancing optimizations. The models also provide greater insight into the reasons why the optimizations work. A compiler can use the models to select profitable optimizations and abandon those that are not effective. It can also learn which applications have similar working set characteristics, and apply predetermined compilation plans that are known to be effective for such applications.

2.6 Implication to memory system design

This thesis introduces a new locality enhancing algorithm, and demonstrates that it can have a significant impact on the design of a memory system. A design space explorer must understand the spatial locality characteristics of an application when evaluating the trade-offs between cache size and block size.

The spatial locality characteristics of a working set dictate the effective cache size that is visible to an application. The amount of pollution (unused data) in the cache increases as spatial locality decreases, and hence the effective cache size decreases. The relation between the actual cache size C and the effective cache size C_e during an interval of execution i is expressed as $C_e = \mathrm{SL}(i, B) \times C$. The effective cache size is hence directly proportional to the spatial locality of a working set.

A locality enhancing optimization that halves the physical working set size of an application leads to a doubling of the spatial locality and effective cache size. The $2\times$ improvement in spatial locality implies that the cache size C may be reduced in half without compromising overall performance. A $2\times$ reduction in the cache size translates to an equal reduction in cache area and power [34], as well as reduction in cost.

Spatial locality was shown to depend on the block size. The exploration process can devise an algorithm to analyze the application working sets and calculate the largest B such that $C_e \approx C$. The search for the largest B is self evident. A larger block amortizes an expensive memory access over more elements. In practice, the block size is constrained in several ways because it impacts the memory and bus architectures. Larger block incur more substantial investments with respect to area, power, and cost. In addition, B is often a power of two, and it is empirically observed that the effective cache size changes negligibly for block sizes larger than 128 *au*s (32 words) (i.e, the ratio of used to unused data within a block does not change significantly). All of these constraints are advantageous because they render the search for an adequate block size computationally tractable and practical.

2.7 Other considerations

Locality enhancing optimizations represent one class of memory optimizations. Another class is embodied by latency hiding optimizations that anticipate future memory references and prefetch them into the fastest memories ahead of their actual use. Prefetching serves to mask the long latency associated with a memory access.

A generic prefetching process is illustrated in Figure 7. When a reference is not found in the cache, the address is snooped by the prefetching engine and analyzed in the context

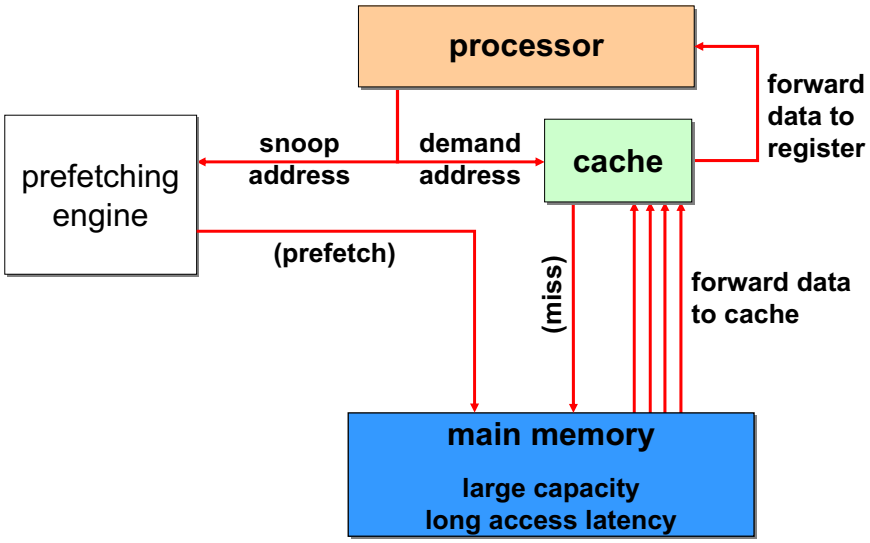


Figure 7: Example prefetching process.

of a local history that the engine maintains to guide its decisions. The prefetcher may then initiate the retrieval of additional data items from memory. The prefetched data are stored in the cache until they are referenced by the processor. An effective prefetch increases processor throughput by reducing or eliminating access latencies.

The block-fetch strategy that is design to exploit spatial locality is a simple prefetching strategy. The lack of spatial locality defeats the block-fetch strategy and leads to great demand bandwidth requirements as Equation 9 shows. There are many forms of prefetching, varying in complexity and efficacy. As with the block-fetch strategy, an ineffective strategy increases the demand bandwidth and reduces the effective cache size. However, unlike the block fetch strategy, it may not reduce the demand bandwidth of an application. This is because a correct prefetch simply shifts references earlier in time, and does not affect the turnover rate between working sets.

The complexity of a prefetching strategy is measured with respect to the amount of state it requires to guide its decisions. For example, a prefetching strategy based on stride predictions [21] needs to store an instruction identifier, the last address referenced by that instruction, a calculated stride, and a confidence measure. In contrast the complexity of a Markov-based prefetcher [13, 21, 26, 47] increases with the complexity of the observed

address sequence. Often its complexity is bound according to heuristics that are observed to work well in practice. These and similar kinds of prefetching strategies usually require non-trivial architecture modifications, and are therefore unattractive for embedded memory systems.

It is possible however to shift the complexity of the prefetching engine into software by allowing a compiler to orchestrate a prefetching strategy. For loop-based computation over array data structures, a compiler can perform data reuse analysis [38] and determine that a prefetch is required in iteration k to mask the latency of a reference in some future iteration $k + l$, where l is proportional to the memory access latency.

This work in this thesis extends the prowess of compiler-orchestrated prefetching to encompass computation that is predominantly characterized by accesses to heap data objects. This contribution facilitates the introduction of cheap and effective prefetching strategies into embedded memory systems.

2.8 Directions for future work

Conceptually, each (intermediate) working set corresponds to some logical partition of the program. For example, consider the application shown in Figure 8 which implements a standard finite impulse response (FIR) filter. The top level function in the program performs some initialization, prints the input signals (two vectors), computes the impulse response, and prints the output signal. In the Figure, the x-axis represents time progressing from left to right. The y-axis represents the values of the program counter. The graph shows the value of the program counter at different times during the execution. Following the graph from left to right, the program starts in the `main` routine, then jumps to the `init` routine, follows by two calls to `print`, a call to `fir`, and finally a call to `print` again.

One might consider a partitioning of the application working set into five intermediate working sets, one for each function call. In general however, an automatic and meaningful partitioning is often impractical because it requires semantic analysis of the computation. A semi-automatic alternative can require user annotations to define likely partitioning boundaries. This thesis defers the issue of automatic partitioning to future work.

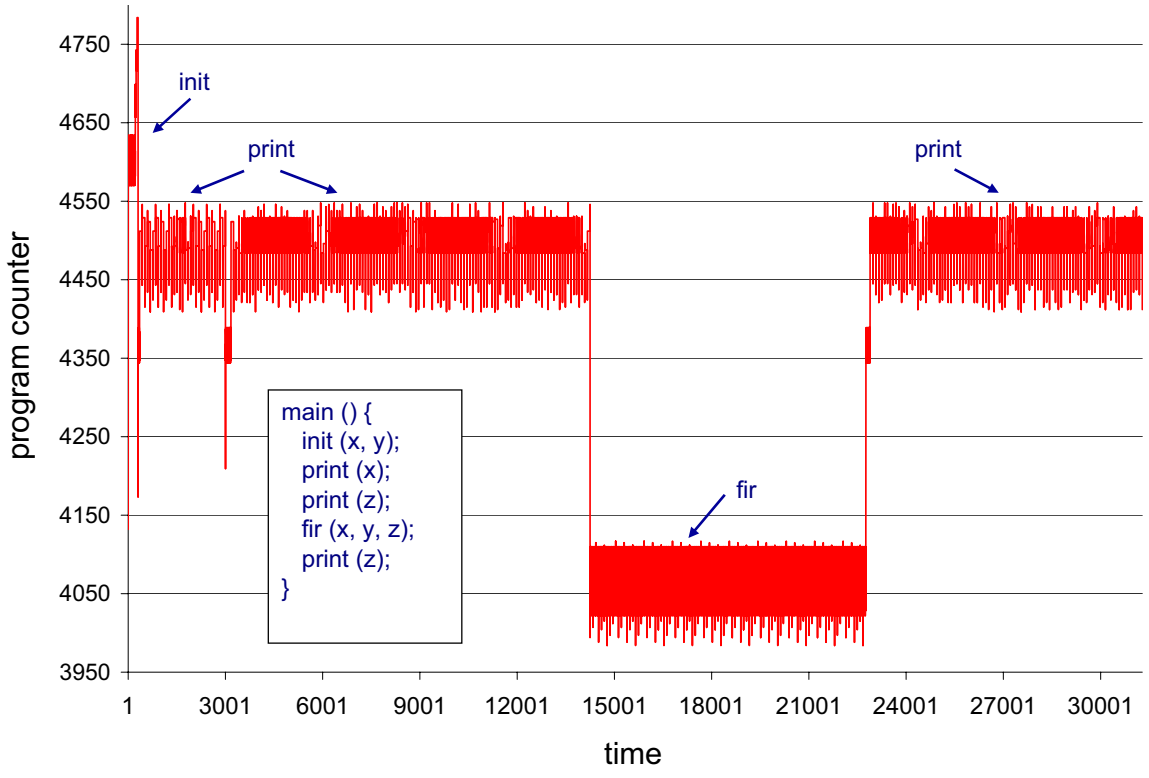


Figure 8: Example control flow trace.

2.9 Summary and remarks

This chapter presents models to quantify well known concepts of temporal and spatial locality. The same models that show how locality impacts performance are also shown to affect memory system design. This is unlike prior work that focused on one dimension or the other [19, 45, 49, 3, 4, 37, 44].

The new models highlight the main insight of this thesis, namely that locality enhancing optimizations can directly optimize the size, power, and cost of the memory system. The analytical framework presented in this chapter serves as a foundation for compiler-guided design of embedded memory systems.

CHAPTER III

DATA REMAPPING

Data structures are fundamental to algorithm design. As applications have evolved, so have the types of data structures they use. Common examples include linked lists, graphs, and trees, although many more exist. A salient attribute of these structures is that they are dynamically allocated, and they are referenced with pointers (a variable whose value is a memory location). This allows programmers to easily build and manage large and complex data structures.

At the programmer disposal are memory allocation routines that are used to reserve locations for the data objects. Usually although not necessarily, there are also routines to free a reserved location so that it may be reused for subsequent allocations. the allocation routines are highly optimized for fast response times, and are designed to be generic. They are thus largely agnostic to the underlying memory hierarchy or the data types allocated. When applications use these allocators the end result is a layout of objects in memory with interconnections that appear irregular and complex. Such object organization is detrimental to performance because it leads to poor spatial locality. Programmers often implement their own custom allocation routines to overcome some of these problems, even though in practice this is not only time consuming, but it is also prone to errors, and poses a challenge for application portability.

For a given record type, different instances of that record (objects) may require different field layouts. A stack object is a record allocated locally to a function and resides on the stack. It often requires a compact layout since it has a short lifetime, and all or most of its fields are used within a short time interval. The fields in stack objects have strong affinity to each other. In contrast, objects that have global scope or those that are dynamically allocated often require alternate layouts because different subsets of their fields are used during different intervals of execution. Furthermore, the fields typically exhibit inter object

affinity in that an access to the fields of one object is followed by accesses to the same fields from other objects of the same type. Different fields are used in different intervals of execution, and hence it is prudent to spatially collocate the fields that are used within close temporal proximity. Such fields may be from the same object (as is the case with stack objects) or from different objects (as is the case with dynamically allocated objects).

Consider for example a function that searches through a linked list of records and replaces a certain data item matching a search key. Each record consists of a `key` field, a `datum` field and a `next` field pointing to the next record in the list. Here, the `key` and `next` fields will be accessed in succession and more frequently (hot fields) than the `datum` field (cold field). Therefore, it would prove beneficial to fetch and cache as many hot fields as possible with each memory access. An allocation strategy that collocates the `key` and `next` fields of various objects in the same memory block, and allocates all the of `datum` fields to a separate block, will improve the program spatial locality which in-turn favorably impacts memory system behavior. Note that packing field-pairs in the same block (i.e., a block containing `key` and `next` fields) does not offer an advantage over individual field packing (i.e., a block containing only `key` fields) since the same number of blocks will eventually be fetched from memory in this case.

Data remapping is an innovative combination of customized placement and field reordering such that the new data layouts exhibit better spatial locality. All previous techniques for data reorganization of dynamically allocated data [10, 15, 31, 42, 55] are characterized by one or more of the following limitations. First, they are not completely transparent to the programmer and require some manual re-tooling of the application. Second, they incur significant run-time overhead as objects are dynamically relocated in memory. Third, they may violate lead to erroneous computation in some applications. In contrast, data remapping is (*i*) completely automated, (*ii*) does not perform any run-time data movements, (*iii*) preserves correctness for a larger scope of applications and (*iv*) is lightweight with a running time linear in the size of the program.

3.1 Remapping with offset manipulation

When an object is dynamically allocated, a pointer is returned to the first addressable unit in the reserved segment of memory. This is the base address of the object. An object is a sequence of fields $f_1, \ldots, f_n$. The address of a field f_i is calculated by adding a fixed offset to base address of the object. The offset of a field equals the sum of all preceding fields in the sequence. Hence, field f_1 has a zero offset, and in general

$$\text{offset}(p \rightarrow f_i) = \sum_{j=1}^{i-1} \text{size}(p \rightarrow f_j) \tag{10}$$

where p is a pointer to an object and $\text{size}(p \rightarrow f_i)$ equals the size of field f_i in addressable units. This same offset function is used for all objects, regardless of their type or access patterns. Some programming languages specify how fields should be organized, while others hide such details from the programmer. The C programming language for example requires that fields are laid out in the order they are declared [29]. This standardization is exploited by programmers who use the presumed layout information to perform their own address calculations for fast data dereferencing.

Data remapping performs locality enhancing optimizations by using different offset calculations for different data types. It does not violate the C language standard, although relaxations of some of the layout specifications would empower greater flexibility in locality optimizations. Figure 9 shows the data remapping impact on spatial locality as defined in the previous chapter. The figure shows the intermediate working sets spanning the lifetime of execution of the `tsp` benchmark [12]. It implements a traveling salesman algorithm which creates a large graph initially, and the iterates over the graph to find the shortest path that covers all the node in the graph. The initialization of the graph appears at the start of execution. It is readily apparent that the physical working set sizes are significantly smaller when remapping is used. The average working set size is halved due to remapping, with many broad peaks virtually eliminated. It is worthy to note that during initialization, the remapped working sets are slightly larger than the original sets. This is because data remapping analyzes the memory access patterns to determine the predominant object

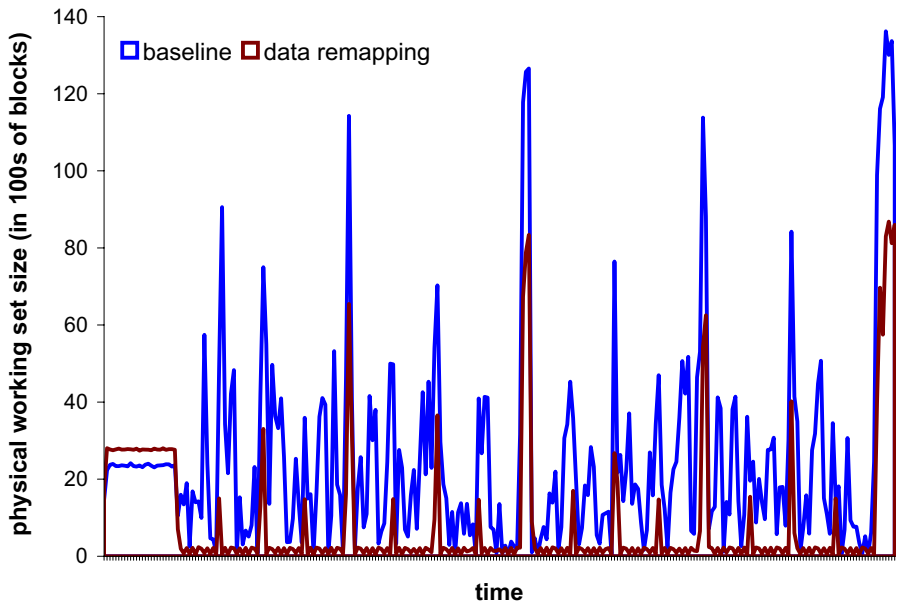


Figure 9: Physical working set size after remapping.

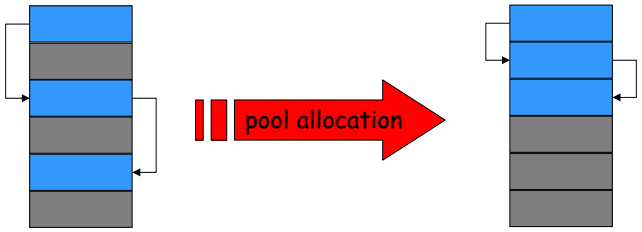


Figure 10: Pool allocation.

affinities. These affinities may vary at different intervals of execution. It is not computationally tractable [43] to simultaneously satisfy every access pattern in the application. Data remapping uses profiling information to heuristically guide the object and field collocation strategies.

Data remapping has two components. The first replaces the allocation requests in the program with custom pool allocators. Each pool is designed to accommodate a fixed number of objects of the same type. Different pools are used for different data types, and new pools are created when existing ones are fully allocated. This pool allocation strategy usually leads to improved locality because same type objects are often used in close temporal proximity of each other. Figure 10 illustrates pool allocation for a linked data structure, and Figure 11

Input: record size r, max field size z, stagger constant S.
Output: Valid base address for new object.

// Cluster, Base and Limit are persistent variables
Initialize *Cluster*, *Base* and *Limit* to 0
if $Base = Limit$ then
    $Cluster \leftarrow$ reserve heap segment of size $S \times r$
    $Base \leftarrow$ base address of *Cluster*
    $Limit \leftarrow Base + S \times z$
end if
$Address \leftarrow Base$
$Base \leftarrow Base + z$
return *Address*

Figure 11: An example pool allocator.

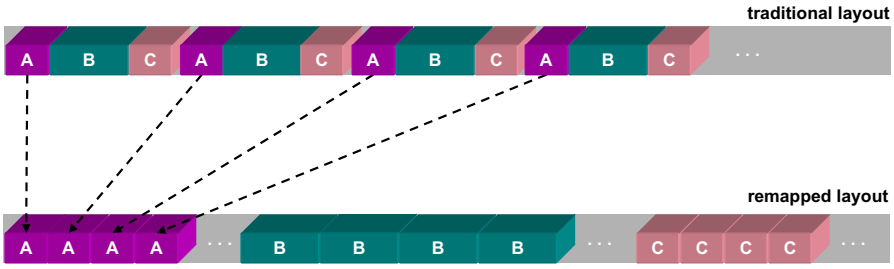


Figure 12: Layout according to different offset calculations.

shows an example implementation of such an allocator.

The second component replaces the standard offset calculation by a custom locality enhancing one. For example, to collocate the same fields from all objects in a pool, the following offset expresses is used

$$\text{remap}(p \rightarrow f_i) = \sum_{j=1}^{i-1} S \times \text{max field size}(p) \tag{11}$$

where S is a static constant that equals the number of objects that can reside in a pool. S is called the stagger constant. It is used in the second term of the equation to ensure that fields do not clobber each other when an object contains a heterogeneous mix of data types. This equation has the same runtime computation complexity as the standard expression shown earlier. The effective layout achieved by the two different offset calculations is illustrated in Figure 12 for a record with three fields `A`, `B`, and `C`.

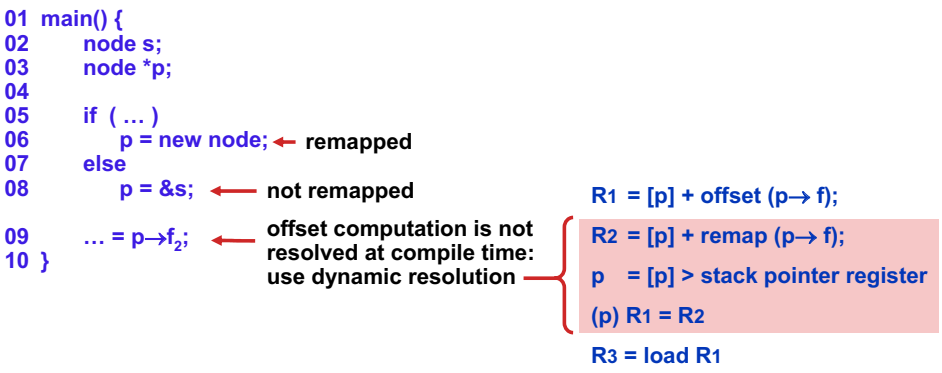


Figure 13: Runtime disambiguation of offset expressions.

Data remapping offers a fast and efficient locality enhancing optimization. The offset expressions may be generalized to express rich collocation strategies. For example, if is possible to collocate multiple fields from the same objects. This is desirable when the fields have strong temporal affinities for one another. It may also be desirable to do so when the largest field in the object is significantly larger than any other field. This reduces the wasted space within a pool.

3.2 Pointers and alias analysis

In its simplest application, data remapping uses two different offset expressions for any given record type. The first is the standard expression shown in Equation 10. It is used for short lived objects that are allocated on the stack. The second is the remapped expression in Equation 11. It is used for dynamically allocated objects.

To ensure program correctness, alias analysis is required to resolve which of the two offset expression to use during code generation. Alias analysis in this context aims to statically characterize pointer accesses into one of three classes: points-to static object, points-to heap object, or point-to unknown. Data remapping handles the class of unknowns using a simple runtime disambiguation strategy that is illustrated in Figured 13. When the compiler is unable to disambiguate a pointer, it evaluates the target address of a field access using both offset expressions. At runtime, it compares the base the address to the stack pointer register. If the base address is found to be a heap object (base pointer is greater than stack

frame pointer for architectures where the stack grows upward and the heap downward), a predicate is set to move the remapped address location into the appropriate source operand.

The implementation of remapping leveraged existing pointer analysis algorithms to reduce the number of runtime disambiguations. In particular, Steensgaard points-to analysis [50] is used since it does not discriminate among the fields of a record. Other types of analysis may be applied if they are found to reduce the number of point-to unknown accesses.

3.3 Remapping of array objects

Data remapping is also applicable to arrays of records. An array of records is traditionally allocated in a contiguous memory segment (cluster) with a statically known starting location (base) and size (rank). The location of a field f_i in an array element $A[k]$ is computed using one of the following two offset expressions

$$\text{offset}(A[k].f_i) = (k-1) \times \text{size}(A[k]) + \sum_{j=1}^{i-1} \text{size}(A[k].f_j) \quad (12)$$

$$\text{remap}(A[k].f_i) = (k-1) \times \text{size}(A[k].f_i) + N \sum_{j=1}^{i-1} \text{size}(A[k].f_j) \quad (13)$$

where $\text{size}(A[k])$ equals the size of the record in addressable units, $\text{size}(A[k].f_i)$ equals the size of the field, and N is rank of the array (i.e., number of elements it contains). The last term in Equation 13 is called the stagger distance. These expressions achieve the same layout shown in Figure 12. The remapping is fully automatic, and performed at compile time for globally allocated arrays. It incurs no runtime overhead. The same remapping strategy is also applicable to dynamically allocated arrays although in this case, the size of the array may not always be resolved at compile time. A dynamically allocated arrays that are remapped incur some runtime overhead as the stagger distance is calculated at runtime. The size of a dynamically allocated array can be associated with its base address and stored in a lookup table.

Input: Object access trace T and window size W.
Output: neighbor affinity probability.

```
01. affinity ← 0
02. for j := W to |T| do
03.     for i := W − 1 down to 1 do
04.         p1 ← T[j]
05.         p2 ← T[j − i]
06.         if (p1 ≠ p2) then
07.             affinity ← affinity + 1
08.         end if
09.     end for
10. end for
11. n ← (W − 1) × (|T| − W + 1)
12. affinity ← affinity/n
13. return affinity
```

Figure 14: Algorithm to compute neighbor affinity.

3.4 Profile analysis

An arbitrary application of the remapping strategy to all data objects in a program does not necessarily increase spatial locality. Some data structures may not exhibit the requisite reference behavior to justify remapping. The profile analysis is geared to discover the most dominant field access patterns. It does not distinguish the relative order of fields in the pattern. This is in contrast to previous work where the temporal behavior of data fields is tracked [14]. Such extensions may enhance the optimization. In addition, since the analysis is profile driven, it is sensitive to the input workload selected for training the optimization.

Data remapping uses the algorithm in Figure 14 to determine if the fields of an object have a strong affinity toward each other, or toward fields of different objects. The measured affinity is called the neighbor affinity probability. It ranges from zero to one, or from low to high inter object temporal affinities. Data remapping is applied to data types have have strong inter object affinities.

The algorithm has a complexity linear in the size of the memory access trace. It processes an object access trace T where all objects occurring in T are of the same type. $\mathrm{T}[i] = p_i$ for $i > 0$, where p_i is a unique object id. Two references p_i and p_j are equal if they have

the same object id. There is a different trace for every record type. The calculated affinity is normalized so that meaningful comparisons can be made. Following the analysis, the compiler marks all record data types with an affinity greater than a chosen threshold for remapping. All other record types are left unmarked and are subsequently organized using the standard memory layout strategies specified by the programming language.

3.5 Limitations

Often large applications use precompiled libraries to reduce development time. If data remapping is partially applied, the modified layout may not be propagated properly to the libraries, and program correctness cannot be preserved. A viable solution is to undo the effects of remapping (data duplication) at a library function call site and reapply the remapping upon return. However, if this is done frequently, the cost of data duplication may not be tolerated and the optimization should be inhibited. In general any library function which operates on objects as a whole poses an issue (e.g., `quicksort`). In embedded systems, it is common to have access to the entire application source code, and thus it is reasonable to assume remapping can be applied throughout the program.

Some dynamically allocated objects do not persist for the entire program lifetime. When a request to deallocate an object is made, a non-empty pool may not be freed. The implemented resolution is to maintain a bit vector per pool to indicate when it may be safely deallocated. In other words, the object deallocation is delayed. This may in some cases result in a larger program memory footprint. Some operating systems actually use a delayed deallocation strategy.

CHAPTER IV

DESIGN SPACE EXPLORATION

Benchmarks from the SPEC [48], DIS[1], and Olden [12] suites were selected for detailed analysis. The Olden benchmarks provide a common frame of reference with previous work on data reorganization [31, 15, 55]. The others provide insight into larger programs. The benchmarks were executed using large reference input sets, whereas profile information was gathered using much smaller training workloads. A short description of the benchmarks is as follows. `164.gzip` is an integer SPEC benchmark. It uses a dynamically allocated array of records during decompression. `179.art` is a floating point benchmark from the SPEC suite. It dynamically allocates an array of records at startup. `dm` and `field` are benchmarks from the DIS suite. The former is a database management system, with many different dynamically allocated objects that are repeatedly updated, deleted and reallocated. The latter uses an array of records that is randomly searched and modified. The remaining benchmarks are provided by the Olden suite. They are memory intensive and allocate substantial amounts of heap objects. The primary data structure used in `health` is a linked list to which elements are added and removed. `perimeter` and `treeadd` respectively allocate quad and binary trees at program start-up and do not subsequently modify them. `tsp` creates a quad-tree at program startup that is repeatedly updated. Additional details for each benchmark, such as the input workload and memory footprint, are tabulated where appropriate (Tables 8 and 3).

Experimental results are presented in two parts. First, data remapping is evaluated as a performance enhancing optimization using commodity processors. Then, in accord with the analytical model presented in Chapter 2, the performance gains are traded to optimize the memory design space.

[1]The Data Intensive Systems (DIS) benchmarks were developed at the Atlantic Aerospace Electronics Corporation, in conjunction with the Boeing Company and ERIM International.

Table 3: Benchmarks, workloads and main memory footprints.

Name	Workload	Memory Footprint
`179.art`	ref1 and ref2	small
`dm`	set14 and set24	24Mb
`field`	11654 and 54860 Tokens	small
`health`	Levels 3-6, Units 1000-10000	123Mb
`perimeter`	11Kx11K and 12Kx12K	147Mb
`treeadd`	20 and 25 Levels	512Mb
`tsp`	3M and 8M Cities	320Mb

Table 4: Data remapping speedup summary.

Processor	CPU Speed	L2 Size	% Speedup
Pentium 3	750 MHz	256 Kb	26
Pentium 2	400 MHz	512 Kb	24
UltraSparc II	400 MHz	2048 Kb	9

4.1 Performance evaluation

The remapping algorithms were implemented in the `GNU` Compiler Collection (`GCC`, version 2.95.2). Profile information was gathered using `Valgrind` [39]. The algorithms were implemented in the compiler front-end, where type information is available and source level transformations are possible. The benchmarks used in this context, as well as the input workloads and memory footprints, are listed in Table 3.

The benchmarks are compiled using two levels of optimization (-O and -O3). Standard `GCC` optimizations are designed to reduce code size and execution time. Aggressive optimizations add function inlining and other techniques that do not involve a time-speed trade-off. Two Pentium and a Sun UltraSparc II processors were used to measure user execution times. The processor configurations and speedup due to remapping are summarized in Table 4.

Table 5 reports the performance results for the Pentium 3 processor. Aggressive (O3) and data remapping (R) optimizations are compared to the baseline (O). For example, the second column reports the percent execution-time speedup when aggressive optimizations (O3) are enabled, relative to the baseline performance (O optimizations in this case). A positive percentage indicates an improvement and a negative percentage is indicative of performance degradation. Similarly, the third column presents the speedup percentage when

Table 5: Results for Pentium 3 processor.

Benchmark	% Execution Speedup			
	O3/O	O+R/O	O3+R/O	O3+R/O3
`179.art`	54.97	36.84	74.72	43.87
`dm`	-17.95	15.12	-3.26	12.45
`field`	78.01	37.67	79.25	5.63
`health`	8.96	36.35	49.13	44.12
`perimeter`	19.10	28.65	46.15	33.44
`treeadd`	12.82	28.02	38.33	29.26
`tsp`	29.28	10.20	37.80	12.04
Average	26.46	27.55	46.02	25.83

Table 6: Results for Pentium 2 processor.

Benchmark	% Execution Speedup			
	O3/O	O+R/O	O3+R/O	O3+R/O3
`179.art`	-9.33	30.94	74.32	76.51
`dm`	-25.39	14.44	-11.84	10.81
`field`	58.05	32.83	45.52	-29.86
`health`	12.03	24.07	39.22	30.91
`perimeter`	21.96	31.91	53.87	40.89
`treeadd`	22.36	10.09	31.05	11.19
`tsp`	2.78	22.69	26.81	24.71
Average	11.78	23.85	36.99	23.59

Table 7: Results for UltraSparc II processor.

Benchmark	% Execution Speedup			
	O3/O	O+R/O	O3+R/O	O3+R/O3
`179.art`	62.03	4.05	61.57	-1.19
`dm`	24.13	-4.67	18.75	-7.09
`field`	86.23	22.08	82.79	-24.98
`health`	16.42	12.80	32.38	19.10
`perimeter`	24.14	25.00	51.72	36.36
`treeadd`	28.89	22.91	50.51	30.39
`tsp`	45.04	1.43	48.89	7.00
Average	40.98	11.94	49.52	8.51

Table 8: Benchmarks for design space exploration.

Name	Workload	Memory Footprint
`164.gzip`	test	15Mb
`179.art`	test	small
`field`	11654 Tokens	small
`health`	8 Levels, 100 Units 100	41Mb
`perimeter`	11Kx11K	146Mb
`tsp`	1M Cities	40Mb
`treeadd`	22 Levels, 20 Iteration	64Mb

baseline optimizations are combined with data remapping (O+R) and compared to baseline optimizations alone. In some cases (`dm`, `health`, `perimeter`, `treeadd`) data remapping alone outperforms aggressive compiler optimizations. Results for the Pentium 2 and UltraSparc are reported in Tables 6 and 7 respectively. Speedups are generally similar although some optimization combinations lead to performance degradation. No attempts were made to determine which compiler optimizations benefit from or are inhibited by data remapping. This is left for future work.

4.2 Design space optimization

While data remapping affords significant reductions in execution time, the objective here is to explore the possibility of trading-off some of the benefits in order to reduce the memory needs of the applications. The experiments that follow were carried out using a detailed processor and memory simulator [54]. The benchmarks that are evaluated appear in Table 8. The benchmarks were completely simulated for various memory hierarchy configurations. The results reported here assume bandwidth of 8 bytes per cycle throughout the hierarchy, 4-way associative caches, and a first-access memory latency of 30 cycles. Memory is configured to provide streaming support for subsequent accesses.

Table 9 summarizes the processor and memory hierarchy performance for a baseline architecture, before remapping. It reports the total execution cycles of the application and the energy consumed in primary cache (L1) and secondary level (L2) caches. Table 10 demonstrates the impact of remapping with respect to energy and performance using the

Table 9: Execution cycles and energy results before remapping.
(L1=32KB, L1 line size=16 bytes, L2=1MB, L2 line size=32 bytes)

Benchmark	Execution Cycles	L1 Cache Energy (J)	L2 Cache Energy (J)	L1+L2 Energy (J)
164.gzip	1106079932	0.0311	0.085	0.116
179.art	704713706	0.465	9.437	9.938
field	1047393960	3.838	0.922	4.76
health	2616712073	1.293	21.489	22.782
perimeter	813958394	1.4189	7.151	8.5699
treeadd	877485849	1.243	4.497	5.731
tsp	1077624556	1.868	8.676	10.544

Table 10: Execution cycles and energy results after remapping (relative to Table 9).
(L1=32KB, L1 line size=16 bytes, L2=1MB, L2 line size=32 bytes)

Benchmark	Execution Cycles	L1 Cache Energy (J)	L2 Cache Energy (J)	L1+L2 Energy (J)	% Reduction in Energy	% Reduction in Execution
164.gzip	1083997353	0.0312	0.0848	0.116	0.000	2.00
179.art	216812141	0.17	2.702	2.873	71.090	69.23
field	1047626423	3.838	0.9219	4.7602	-0.004	0.00
health	2044289648	1.263	14.81	16.072	29.337	21.88
perimeter	628221147	1.4178	5.139	6.557	23.488	22.82
treeadd	785358662	1.0344	3.3129	4.347	24.149	10.50
tsp	926038088	2.245	4.985	7.23	31.430	14.07
Average	–	–	–	–	25.640	20.07

Table 11: Execution cycles and energy results after remapping (relative to Table 9).
(L1=32KB, L1 line size=16 bytes, L2=512KB, L2 line size=32 bytes)

Benchmark	Execution Cycles	L1 Cache Energy (J)	L2 Cache Energy (J)	L1+L2 Energy (J)	% Reduction in Energy	% Reduction in Execution
164.gzip	1083997353	0.0312	0.0439	0.0751	35.258	2.00
179.art	250995040	0.17	1.418	1.588	84.021	64.38
field	1047626423	3.838	0.475	4.313	9.391	-0.02
health	2044289648	1.26	7.625	8.888	60.987	21.88
perimeter	628311657	1.4178	2.6489	4.0667	52.547	22.81
treeadd	785407236	1.0344	1.7059	2.74	52.189	10.49
tsp	956363728	2.245	2.6103	4.855	53.955	11.25
Average	–	–	–	–	49.764	18.97

Table 12: Execution cycles and energy results after remapping (relative to Table 9).
(L1=16KB, L1 line size=16 bytes, L2=512KB, L2 line size=32 bytes)

Benchmark	Execution Cycles	L1 Cache Energy (J)	L2 Cache Energy (J)	L1+L2 Energy (J)	% Reduction in Energy	% Reduction in Execution
164.gzip	1113402555	0.02	0.0488	0.0689	40.603	-0.66
179.art	251159762	0.1078	1.4177	1.525	84.654	64.34
field	1047626417	2.474	0.474	2.949	38.046	-0.02
health	2046953548	0.8014	7.7112	8.513	62.633	21.77
perimeter	628440207	0.9088	2.6526	3.561	58.448	22.80
treeadd	785670341	0.6623	1.713	2.375	58.559	10.46
tsp	975110313	1.446	3.083	4.529	57.046	14.31
Average	–	–	–	–	57.141	19.00

original hardware configuration. Table 11 reports the energy and performance results after remapping but for a secondary cache of half the original capacity. Finally, Table 12 demonstrates the impact of remapping on energy and execution time when the primary and secondary caches are half their original size.

The energy consumed in the caches is modeled according to the approach of Kamble and Ghose [27]. The performance results (cache hits and misses) from the simulation are combined with the cache parameters (cache capacity, line size and tag size) to derive memory signal transition counts. The model requires capacitance parameters, such as the metal wire capacitances as well as the gate and drain capacitances of the transistors in different parts of an SRAM circuit. These are calculated following the example of Wilton and Jouppi [57]. TSMC 0.25μ technology parameters are used to simulate the corresponding components of an SRAM circuit in HSpice. Further details are contained in a technical report [34]. One drawback to the model is that it does not model I/O pads. A more important drawback is that the model only accounts for dynamic power dissipation, which makes it inaccurate for smaller geometries (0.09 μ technology) with relatively large static (leakage) power dissipation. The model is reasonably accurate for the 0.25μ technology assumed here. In this regime dynamic power dissipation is approximately two orders of magnitude greater than static power dissipation [52].

The largest energy reduction always occurs in the smallest cache configuration (Table 12). With half sized caches, the number of cache entries is halved, thus halving the capacitance seen on any particular bit line. Since power is proportional to the capacitance and voltage (CV^2), halving the former capacitance halves the power consumed. Also note that after remapping, the primary cache energy tends to increase. This is due to more L1 cache hits. However, the overall energy of both L1 and L2 is significantly reduced, due to the greatly decreased number of cache lines exchanged between L1 and L2.

The largest execution time reduction most often occurs using the original cache configurations (Table 10). This is in accord with the analytical model that suggests performance improves as the cache size increases and the working set size decreases (see Table 1).

CHAPTER V

INTROSPECTIVE PREFETCHING

Latency hiding optimizations anticipate future memory references and prefetch them into the fastest memories ahead of their actual use. Prefetching attempts to mask long memory access latencies. There are many techniques designed to either statically [11, 32, 38, 40, 58] or dynamically [46, 13, 21, 26, 47] predict and prefetch memory references. Static schemes generally require little architecture investments beyond an instruction set architecture (ISA) that supports prefetching. On the other hand, dynamic schemes often require non-trivial architecture modifications. Predictive strategies generally work well for array-based computations, but they are vulnerable to the irregular memory access patterns that are common to pointer-heavy applications.

There are prefetching strategies that are better suited for pointer-heavy applications. They redundantly execute subsets of an application to provide the information necessary to perform data prefetching. These techniques are thus guided by precomputation that is often effective in the timely discovery of future memory references, but at a substantial cost in architectural investments that can include dedicated micro-engines, threads, or even co-processors [6, 16, 17, 30, 35, 36, 56, 59].

The architectural investments associated with existing precomputation techniques cast serious challenges on making them practical to embedded processors. Hence it is desirable to implement a prefetching strategy capable of substantial foresight about future memory reference patterns, but with the architectural complexity of static prediction techniques. This chapter addresses this challenge with introspective prefetching, an innovative combination of speculative and predicated execution.

Introspective prefetching is a unified compiler-orchestrated strategy for array and pointer heavy applications. It uses cache-miss profiling to focus on a relatively small set of instructions that suffer from poor cache performance. The compiler then automatically embeds

Table 13: Delinquent loads in different benchmarks.

Benchmark	Total number of static loads	Number of delinquent loads
130.li	1820	18
132.ijpeg	5079	43
164.gzip	1226	9
175.vpr	5289	30
181.mcf	515	14
183.equake	945	30
188.ammp	776	3
255.vortex	21298	361
256.bzip2	1064	28
300.twolf	10695	99

new instructions into the host program to orchestrate runtime precomputation and prefetching. The new instructions execute speculatively as part of the same instruction stream as their host. They effectively run ahead to carry out data prefetching. A significant aspect of this work is its ability to dynamically adapt to runtime information and dynamic behavior. For example, the precomputation instructions can self-nullify when they are likely to increase the burden on the memory system.

Introspective prefetching is designed to increase the synergy between static prefetch orchestration and run-time adaptation in response to information propagated from the memory system. It requires a mechanism to query if an address is in the cache, and uses the result of the query to dynamically throttle the precomputation, akin to informing memory operations [23] that invoke special handling routines upon a cache miss. Conceptually, the result of the query is a predicate that guards the execution of precomputation instructions. The query mechanism can be readily implemented in modern architectures, and exposed to the compiler via the ISA. It is otherwise readily applicable to existing embedded processors and other commodity architectures.

The remainder of this chapter illustrates an example and describes the compiler details in greater detail. This is followed by architecture considerations and an empirical evaluation.

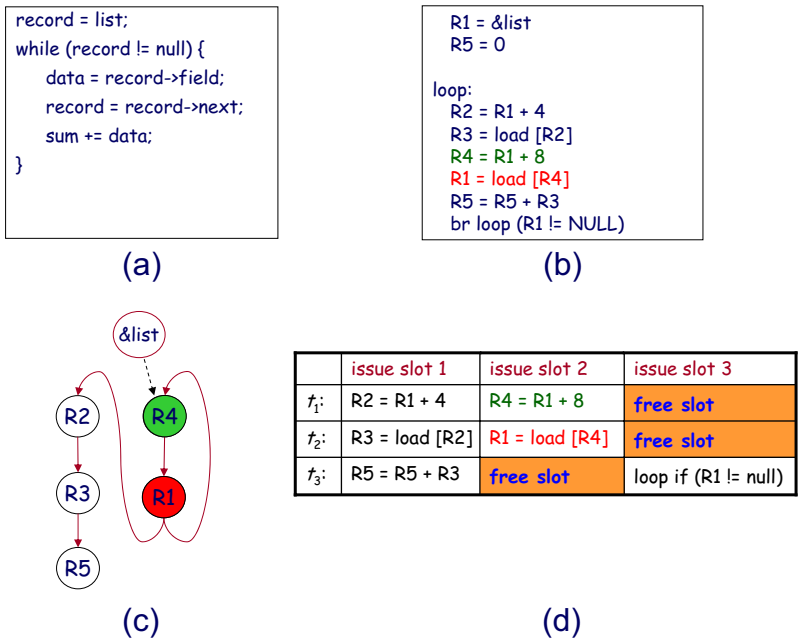
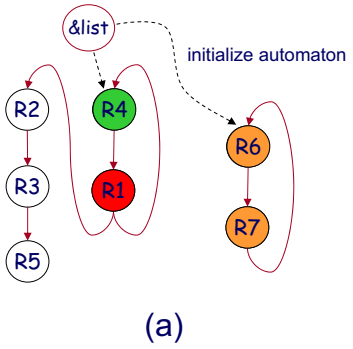


Figure 15: Example pointer-chasing code.

5.1 Introspective prefetching example

The analytical model presented earlier in the thesis shows that prefetching must be done judiciously to avoid polluting the cache and reducing the effective bandwidth. Previous work [2, 1, 41] has shown that a small number of load instructions are delinquent, accounting for more than 90% of the total cache misses that a program suffers. Table 5.1 shows the number of such loads in for programs commonly used for benchmarking (from the SPEC CPU benchmark suites). The compiler leverages this insight to focus the optimization. It can use profiling information to identify delinquent loads and apply introspective prefetching only when it is most profitable.

An example code snippet is shown in Figure 15(a). It is a pointer chasing loop that traverses a linked list of records. The delinquent load corresponds to the statements responsible for the retrieval of the next record in the list (`record = record->next`). In the low level representation of the loop listed in (b), the delinquent load is the fourth instruction in the loop (`R1 = load [R4]`). A potential schedule of execution in shown in (d). It assumes a three issue processor and no software pipelining.



	issue slot 1	issue slot 2	issue slot 3
t_1:	R2 = R1 + 4	R4 = R1 + 8	**R7 = load [R6]**
t_2:	R3 = load [R2]	R1 = load [R4]	**R6 = R7 + 8**
t_3:	R5 = R5 + R3	**Prefetch [R6]**	loop if (R1 != NULL)

(b)

Figure 16: Example precomputation and prefetching.

The compiler can recognize that there are available scheduling slots that may be used to precompute future addresses and prefetch data. A driving insight is that a processor can carry out a significant amount of computation to avoid a main memory reference. For a 1-3 GHz processor, the number of cycles to access DRAM currently range from 100 to 300 cycles, or more. The latency can increase significantly higher to 500-1000 cycles when queuing delays are accounted for in emerging multicore and tiled processors. Accordingly, processors spend 10-20 cycles in the cache hierarchy before accessing external memory. A compiler can leverage the available slots to execute new instructions that precompute and prefetch future memory references.

The precomputation is guided by the program itself. A compiler can analyze the code to identify a program slice [7, 53] of the set of operations that contribute to the address calculation of a delinquent load. Each slice is called a load dependence chain (LDC), or equivalently, a precomputation chain. In the example, the LDC is readily apparent from analyzing the flow of data between registers (Figure 15(c)). The compiler can identify the LDC and clone all of its instructions. They are in turn scheduled in the available issue slots (Figured 16) such that now as the program executes, the cloned instructions execute concurrently. They serve to improve performance by hiding long memory access latencies.

The compiler can orchestrate the prefetching such that data prefetched in one iteration of a loop arrives in time for processing in a future iteration. This is achieved by unrolling the LDC as shown in Figure 17. The unrolling is a lightweight transformation that entails

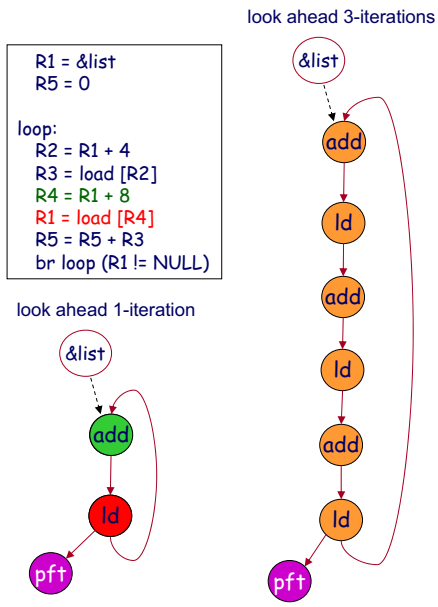


Figure 17: Example LDC unrolling.

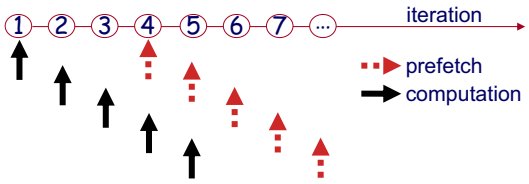


Figure 18: Time line for a precomputation distance of three iterations.

creating γ copies of the LDC and chaining them all together. The unroll factor γ is equal to $\lceil L_l/K \rceil$ where L_l represents the average miss latency of a delinquent load l, and K represents the length of the longest path in the loop region. An example time line is illustrated in Figure 18. It shows that when the precomputation is in iteration i, the precomputation is prefetching data $i + 3$, for a lookahead distance of three iterations.

Cyclic dependences such as the one shown in Figure 15(c) permeate the load dependence chains in pointer heavy codes. They pose a significant challenge to prefetching because the results from each of the loads are necessary to further propagate the precomputation. Herein lies the challenge. When a precomputation load suffers a cache miss that is not serviced in time for a dependent instruction, the processor stalls until the miss is serviced. This is not desirable since the precomputation can adversely impact performance.

This work addresses the challenge with introspective instructions. An introspective load can query the contents of the cache and determine if it is profitable to execute its dependent instructions. It can cancel its dependents by communicating a guarding predicate whose value reflects the outcome of the query. The process is illustrated in Figure 19. An introspective load (appears as `ild` in the figure) queries the cache for an address. If the

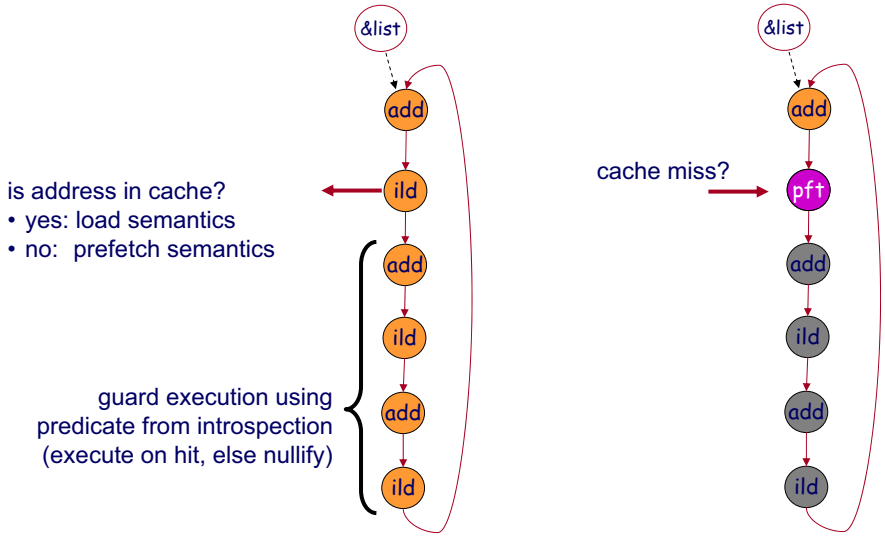


Figure 19: Introspective execution.

address is found in the cache, the instruction executes as a normal load binding the value from the cache to a register. This value is used by its dependent instructions. In the case of a cache miss, the introspective load changes its semantics. It operates as a prefetch for the current address. It also invalidates its dependents so that they do not execute. This avoids stalling the processor. It also affords the unique advantage of dynamically determining which of the future references is most worthy of a prefetch. The canceled instructions resume execution at a later time when it is profitable to do so.

5.2 Implementation details

There is a simple algorithm for identifying load dependence chains. The input to the algorithm is an already optimized and scheduled program expressed in a low level representation. The algorithm also inputs the memory profiling information to identify delinquent loads. The output is a program with embedded precomputation and prefetching instructions. The algorithm avoids the complexities attributed to slicing by leveraging the following program properties:

- Each function consists of a set of blocks or regions. Each block has single entry instruction, and it is the first operation in the block.

- Each operation o in a block is a member of a unique instruction word or bundle w_o. The bundles are assumed to issue in order.

- Instructions within the same bundle are assumed not to have any dataflow dependencies between them.

- The schedule time of a bundle w is $t(w)$, and hence the schedule time of an operation o is $t(w_o)$. Within a block, each bundle has a unique schedule time.

The algorithm identifies delinquent loads as those responsible for 90% or more of the total program misses. For every load l in the set, the algorithm considers each operation p occurring in the bundles preceding w_l, and if there exists a dataflow edge from p to l, a speculative version of p is added to the LDC of l. The LDC is maintained as a queue with operations inserted at the head to preserve data dependencies. When the algorithm encounters a block entry instruction, it may cross the region boundary to continue building the LDC. If a block has multiple (control) predecessors, the LDC is replicated, and for each predecessor, a path-specific LDC is built. For each LDC, the algorithm uses branch profiling to restrict the extraction to the two most frequently traversed paths. This threshold is selected empirically. It serves to limit the number of LDCs that may otherwise arise. The path that is traversed during the LDC construction defines the set of hosts within which the precomputation chain is later scheduled.

The algorithm terminates a precomputation chain according to several heuristics. One of the stopping conditions is triggered when there are no remaining dataflow edges to process. Another terminating condition is triggered when the length of the LDC reaches a predefined limit. The length constraint throttles the amount of precomputation. An experimentally determined threshold of seven instructions is used although the unrolling of the LDC can lengthen the precomputation. A third stopping condition is triggered when a dataflow cycle is detected within the LDC. Cycles occur when the result of a delinquent load impacts its future address computation. This type of recurrence is common to pointer-chasing applications (see Figure 15 for an example).

When the algorithm stops building a precomputation chain, a scheduler begins the process of assigning LDC instructions to available resource slots within the appropriate host regions. Instructions are first injected in the block that is visited last. When all of the scheduling slots in that block are exhausted, the scheduler searches for additional resources in neighboring hosts to avoid lengthening the program schedule. In loop regions, a visit to neighboring blocks may result in traversing the loop back edge. It is therefore possible that scheduling begins in the tail end of a block, and continues at the head of the loop.

The scheduler begins with the first instruction in the LDC. When the LDC is cyclic, the chain is reversed and then scheduled. As each LDC operation is assigned to a resource slot, its destination operand is renamed and register allocated. The scheduler maintains a mapping of the old operand names to the new names, and propagates the new operands throughout the precomputation chain. If the precomputation contains a load instruction, it is converted to an introspective load, and subsequent instructions are properly predicated.

5.3 Introspective execution

Introspective instructions allow the precomputation to adapt in response to dynamic memory events. Specifically, the execution of the precomputation chain is predicated on special-purpose bits of information that signal when certain memory requests are outstanding. The special-purpose bits are akin to event registers that might indicate a cache miss, or the "operand not ready bit" in conventional processors. An introspective memory operations require an extension to the instruction set architecture (ISA).

The assembly language syntax for an introspective load is the same as a standard predicated load instruction: `(p) ild rd = [rs]` where `p` is a guarding predicate, `rd` is the name of a destination register, and `rs` is a the source operand. The instruction has the following semantics:

- The execution of the operation is guarded by a predicate (`p`). When the predicate is cleared (`p` = 0) the operation is nullified (i.e., the state of the machine does not change). Otherwise, the introspective load executes as follows when the predicate is affirmed (`p` = 1).

- If the address in `rs` hits in the TLB and in the primary data cache, then `ild` behaves as a standard load instruction. A value is read from the address specified by register `rs` and placed in register `rd`.

- Otherwise, the `ild` behaves as a non-binding prefetch instruction. The line containing the address specified by the value in register `rs` is moved to the highest level of the data memory hierarchy. In addition an introspective bit `ib` is cleared (i.e., set to FALSE as in `ib` = 0).

The introspective bit is a predicate register that is updated based on the result of the introspection. It is used to guard subsequent operations. The following demonstrates introspective prefetching. Consider the following LDC instruction sequence:

```
ld  r1 = [r0]        # cache miss
...
add r1 = r1, 4       # processor stalls
ld  r2 = [r1]
...
```

When the first load suffers a cache miss, an in-order processor will stall if the memory request is outstanding when the `add` operation is issued. In contrast, the processor can ignore the operation when the introspection suggests the pipeline will unduly stall because of a cache miss:

```
ild r1 = [r0]         # cache miss, ib = 0
...
(ib)  add r1 = r1, 4  # operation ignored
(ib)  ild r2 = [r1]   # operation ignored
...
```

In the above sequence, the processor ignores the embedded precomputation when the introspective bit is not affirmed. Thus when the first introspective load results in a miss, the introspective bit is cleared and subsequent operations in the LDC are canceled. This

scheme allows the precomputation to greedily proceed along, and affords a mechanism for prefetching data for any one of multiple loads in the LDC.

5.4 Architecture considerations

The addition of a new predicate register in the form of the introspective bit is straightforward. The introspective bit may be implemented as global predicate registers, or as an implicit argument to introspective instructions. Either approach conserves the address space of predicate registers, although in each case, an explicit clearing instruction is necessary to reaffirm the introspective bit for future iterations. Naturally, aliasing effects can occur if there are overlapping LDCs in the same program region. That is, the precomputation of one LDC may cancel the precomputation of another even though the two LDCs are distinct and do not share data. Empirical analysis indicates that two precomputation chains often overlap, whereas three or more LDCs in the same program region rarely occur. Hence using two distinct introspective bits (that are exposed to the compiler) can reduce aliasing effects.

An alternate approach employs an introspective bit per register, and in order to reduce the associated architectural complexity, an ISA may dedicate a subset of its register file for introspective prefetching. In this approach, the mechanism to set and clear the introspective bits is similar to the propagation of exceptions flags with NaT bits in EPIC architectures [28, 24]. When an introspective load is issued, the introspective bit associated with its destination operand is cleared. The bit is later set when the memory request is serviced, and in general, the bit is reaffirmed whenever a value is written to a register. A subsequent instruction is canceled if any of the its source operands have a cleared introspective bits. When the instruction is suppressed, the introspective bit of its destination register is cleared to propagate the cancellation of the precomputation chain.

5.5 Memory system performance

Several benchmarks were used for the evaluation of introspective prefetching. The benchmarks and their input workloads are reported in Table 5.5. They are grouped into two categories. The first represents the array-heavy benchmarks drawn from SPEC CFP, and

Table 14: Benchmarks and input workloads.

Benchmark	Profile	Evaluate
101.tomcatv	train	lgred
168.wupwise	train	lgred
171.swim	test	train
172.mgrid	train	lgred
188.ammp	train	lgred
130.li	train	au, boyer, puzzle0
132.ijpeg	vigo.ppm	penguin.ppm
164.gzip	input.random	lgred.graphic
181.mcf	train	lgred
256.bzip2	input.random	lgred.graphic
300.twolf	train	lgred

the second represents pointer-chasing benchmarks drawn from SPEC CINT. The largest set of benchmarks is used, limited only by what the infrastructure can successfully compile and simulate.

5.5.1 Methodology

Trimaran is a publicly available compilation and simulation framework for VLIW research based on the HPL-PD [28] architecture. HPL-PD is a parametric architecture in that it admits machines of different composition and scale, especially with respect to the amount of parallelism offered. The HPL-PD parameter space includes the number and types of functional units, the composition of the register files, operation latencies and descriptors that specify when operands may be read and written, instruction formats, and resource usage behavior of each operation. The architecture instruction set is similar to a RISC load-store architecture, with standard arithmetic and memory operations. It also supports speculative and predicated execution. The ISA was extended to include introspective loads, and simulator was extended to implement the necessary semantics.

The Trimaran compiler only provides a C front-end. Fortran SPEC CFP benchmark are converted to C using `f2c`. The compiler includes a number of classic and high level optimizations such as loop unrolling, copy propagation, common subexpression elimination, dead code elimination, aggressive register allocation, and software pipelining, all of

Table 15: Simulated processor.

Functional Units	4 INT units, 2 FP units, 3 BRANCH units, 2 MEMORY units
Register Files	128 INT registers, 128 FP registers, 64 PREDICATE registers, 8 BRANCH registers
Caches	TLB (split) : 8 Kb I and D fully-associative, 32 bytes per line 8 cycles miss latency
	L1 (split) : 32 Kb I and D caches 4-way, 32 bytes per line 2 cycles hit latency (D cache)
	L2 (unified) : 1 Mb cache 4-way, 64 bytes per line 10 cycles hit latency, 200 cycles miss latency

which were enabled for baseline performance measurements. The compiler was extended to implement the introspective prefetching algorithm.

The simulation environment consists of a cycle accurate in-order HPL-PD simulator coupled with the Dinero IV [20] cache simulator. The processor is configured as shown in Table 5.5.1. In addition, there is a BTB, and a two-level gshare branch predictor. The simulator models contention in the memory system and performs cycle-accurate accounting of processor stalls. A stall-on-use model, meaning the processor pipeline only stalls when an instruction is ready to issue but its source operands are not yet available (due to an outstanding memory request). Additional instructions are not issued when the processor is stalled. Processing eventually resumes when the outstanding data items reach their destination. The simulated caches are non-blocking, and the memory units are pipelined such that they can each process a new memory request every cycle.

The MinneSPEC [33] reduced input workloads are used for some of the benchmarks in the evaluation suite. The reduced workloads reduce simulation time to reasonable durations. The MinneSPEC workloads are distributed with Version 1.2 and higher of the SPEC CPU 2000 benchmark suite. The results that follow were obtained by fully simulating each of the benchmarks.

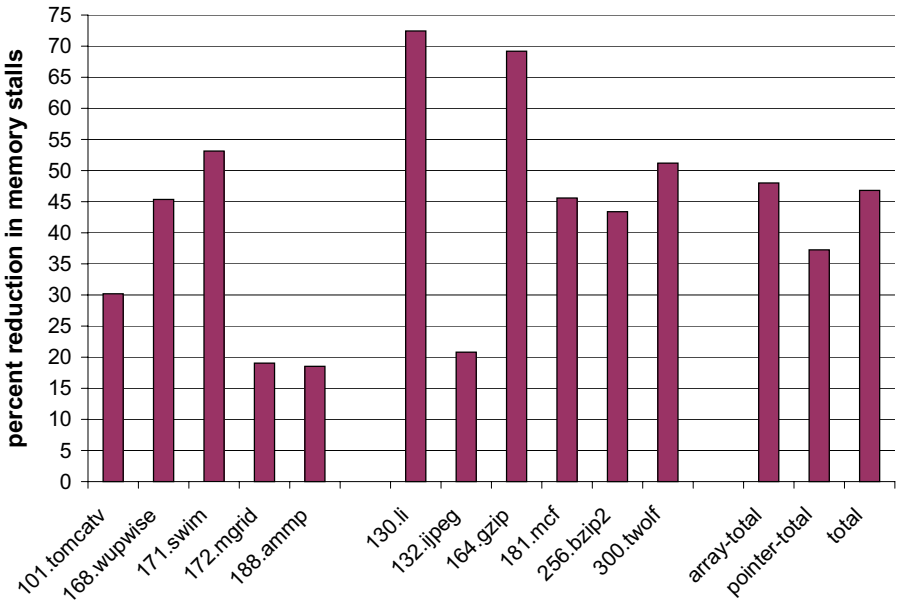


Figure 20: Percent reduction in memory stalls.

5.5.2 Evaluation

Figure 20 reports the percent reduction in memory stall cycles due to introspective prefetching. The performance improves as the percent reduction in stall increases. The results are aggregated in three categories. The first group (i.e., five sets of bars on the left side of the graph) consists of the array-heavy SPEC CFP benchmarks. The second group (i.e., the middle six bars) consists of the pointer-heavy SPEC CINT benchmarks. The last three sets of bars represent the percent reduction in total stall cycles for the array-based (`array-total`) and the pointer-based (`pointer-total`) categories, as well as the overall reduction in stall cycles for all of the benchmarks (`total`). The impact on the memory system performance is significant, up to 70% in the best case. It is 37% for the pointer codes, and 48% for the array codes.

It is possible to isolate the effects of the introspection by altering the compiler strategy such that it does not use introspective loads. Figure 21 illustrates the process for the code snippet shown earlier in Figure 15. The LDC consists of the instructions shown in bold. In Figure 21(b), the scheduler has assigned the first LDC instruction to bundle w_5 and the second LDC instruction to bundle w_1 at the top of the loop. The example

```
    R1 = &list
    R5 = 0
loop:
    w1: R2 = R1 + 4
    w2: R3 = load[R2]
    w3: R4 = R1 + 8
    w4: R1 = load[R4]        # delinquent load
    w5: R5 = R5 + R3
    w6: br loop (R1 != NULL)
```

The load in w_4 is delinquent. Its LDC is:

```
    p2: R1 = load[R4]      # second LDC operation
    p1: R4 = R1 + 8        # first LDC operation
```

(a) Original code

```
    R1 = &list
    R5 = 0
    R6 = R1 + 8
loop:
    w1: R2 = R1 + 4;          p2: R7 = load[R6]
    w2: R3 = load[R2]
    w3: R4 = R1 + 8
    w4: R1 = load[R4]
    w5: R5 = R5 + R3;         p1: R6 = R1 + 8
    w6: br loop (R1 != NULL)
```

The LDC operations are scheduled and the destination registers renamed. This example does not propagate the original cyclic dependence in the LDC (i.e., R7 is not used in p_1). Hence p_2 can serve as a prefetch instruction.

(b) Original code with an embedded LDC

```
    R1 = &list
    R5 = 0
    R6 = R1 + 8
loop:
    w1: R2 = R1 + 4;     p2: R7 = load[R6]
    w2: R3 = load[R2];   p3: R8 = R7 + 8
    w3: R4 = R1 + 8;     p4: R9 = load[R8]
    w4: R1 = load[R4]
    w5: R5 = R5 + R3;    p1: R6 = R9 + 8
    w6: br loop (R1 != NULL)
```

The LDC is unrolled twice and scheduled with register renaming. This example propagates the cyclic dependence (i.e., R9 is used in p_1).

(c) Unrolled precomputation with cyclic dependence

```
    R1 = &list
    R5 = 0
    R6 = R1 + 8
loop:
    w1: R2 = R1 + 4;     p2: R7 = load[R6]
    w2: R3 = load[R2];   p3: R8 = R7 + 8
    w3: R4 = R1 + 8;     p4: prefetch[R8]
    w4: R1 = load[R4]
    w5: R5 = R5 + R3;    p1: R6 = R1 + 8
    w6: br loop (R1 != NULL)
```

This example does not propagate the cyclic dependence. It converts the last LDC operation to a prefetch instruction.

(d) Unrolled precomputation without a cyclic dependence

Figure 21: Example LDC prefetching without introspective instructions.

assumes the processor can issue two instructions per bundle, and a semicolon is used to separate instructions within the same bundle. When the first LDC instruction is scheduled, it is assigned a new register (R6). The source operand of its dependent instruction is updated to reflect the change. The renaming step assures that registers that are used for precomputation do not affect the host program, as noted earlier. In the case of acyclic LDCs an appropriate prefetch operation is added following the last of the precomputation instructions. Cyclic LDC dependences are broken by converting the last instruction in the LDC to a prefetch instruction. This strategy avoids recurrences but will still suffer potential stalls if the LDC has two or more loads, or if the LDC was unrolled as shown in Figure 21(d). The precomputation will prefetch objects i and $i+1$ with every iteration of the loop, and thus maintains a fixed lookahead distance at the cost of prefetching redundancy. Furthermore, the compiler does not need to properly initialize the live-in values to the LDC because the

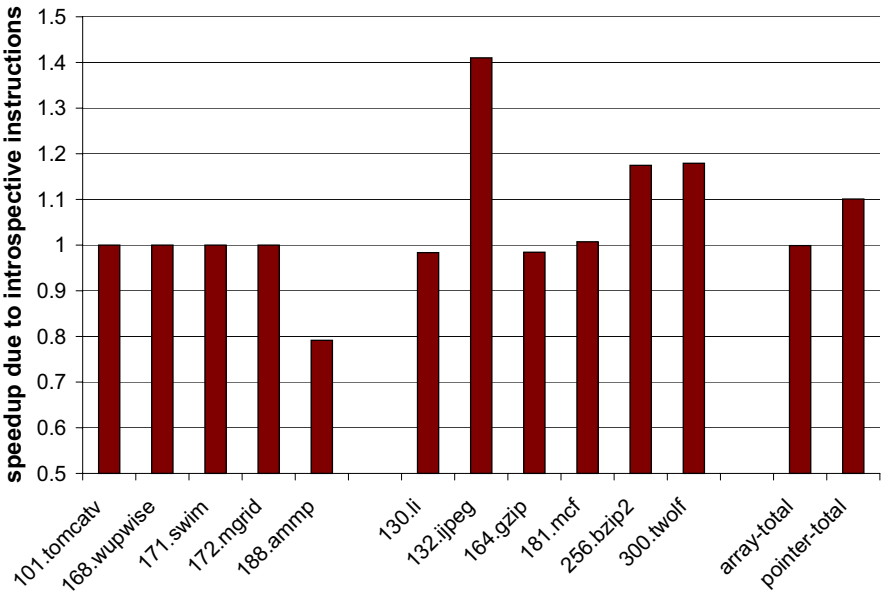


Figure 22: Speedup due to introspective instructions.

precomputation synchronizes with the appropriate values calculated in the original loop at every iteration. The example in Figure 21(c) preserves the cyclic LDC dependence for contrast. This strategy is not evaluated here. The result of the last instruction (`R9`) is consumed by the first operation in the chain (in bundle w_5). Thus iteration i of the loop will access the ith object in the linked list while the precomputation accesses objects $(2 \times i) + 1$ and $(2 \times i) + 2$. The precomputation can run farther ahead when cyclic dependences are preserved.

Figure 22 summarizes the benefits of introspective instructions. As expected, introspective loads do not offer an advantage in array-based codes. For pointer-codes however, the gains are 41% in the best case. The arithmetic mean gain is 10%. The better performance is due to the behavior of load operation within the computation chain. If the LDC contains a load operation q whose average miss latency L_q is less than the miss latency L_l of the target delinquent load, introspective loads that nullify the precomputation are at a disadvantage. This is because if the precomputation waits for the results of load q, the potential reward is L_l, for a total positive gain of $L_l - L_q$. If on the other hand, the average miss latency of the delinquent load is less than some other load within the LDC, introspective loads afford

a exclusive advantage since they will cancel the remaining precomputation. Concomitantly, the outstanding memory transaction becomes a prefetch request. This phenomenon arises when the load dependence chain has more than one load operation, and suggests there are opportunities for selectively using introspective loads in precomputation.

5.5.3 Precomputation overhead

The prefetching strategy was found to increase in the number of dynamic bundles by an average of 3.66%. The overhead is measured with respect to the total number of dynamically executed bundles before and after prefetch orchestration. Each instruction bundle consists of a set of operations that issue simultaneously and execute in parallel. The extent to which the prefetch orchestration lengthens the program is reflected in the number of bundles that are processed. When the compiler embeds the LDCs in regions with an adequate number of available resources, the dynamic bundle count will not change relative to the baseline. The complier used the following heuristic to curb the overhead. During scheduling, it compares the length of the precomputation chain to the estimated number of available resource slots in the host regions. It discards a load dependence chain if its computational requirements are likely to exceed the available resources. The precomputation overhead increases to 32% when this heuristic is not used.

5.6 Summary and concluding remarks

This chapter introduces the concept of introspective prefetching. It is a compiler-orchestrated strategy that uses an innovative combination of speculative and predicated execution. The prefetching is precomputation based. It is focused on delinquent load operations that are most likely to benefit from data prefetching. Introspective prefetching is shown to be an effective unified strategy for array and pointer heavy codes. It requires little change to existing instruction set architectures, and a variant of the strategy is readily applicable to existing embedded processors.

CHAPTER VI

CONCLUDING REMARKS

The memory hierarchy has served as a central component in computing platforms since the introduction of the von Neumann machine. In embedded systems however, the cost of the memory hierarchy limits its ability to play as central a role. This is due to stringent area, power, and cost constraints that fundamentally impact design choices, and limit the physical size and complexity of the memory system.

The emergence of multicore processors, and heterogeneous systems on a chip exacerbates the need for effective methodologies for the design and optimization of memory systems. In contrast to the design of custom computing cores, the design of a supporting memory system remains ad hoc, often relying on intuition and extensive simulations.

This thesis introduced an analytical foundation to characterize intrinsic program properties, and reasons about their implication to memory system design. The thesis also introduced new compiler optimizations that can impact the design choices significantly, and demonstrated that a compiler can play a central role in the design optimization of memory systems. An automated framework which navigates the design space remains a topic for future research. It is conceivable that the exploration can reduce to minimizing linear functions of a finite number of constraints imposed by the analytical framework presented in this thesis.

At the heart of this work is a compile time data remapping algorithm that is designed to enhance locality for dynamic programs with irregular memory access patterns. Stated in simple terms, remapping is a reorganization of the application data in memory, such that memory elements that are accessed contemporaneously are in fact placed together in memory. Thus, remapping aims to improve the spatial locality of memory elements that in fact also share temporal locality.

This thesis also introduced the concept of introspective prefetching as a unified compiler-orchestrated strategy for effective latency hiding techniques. It is an innovative combination of speculative and predicated execution. It is designed to increase the synergy between static prefetch orchestration and run-time adaptation in response to information propagated from the memory system. This contribution facilitates the introduction of effective prefetching strategies into embedded memory systems.

REFERENCES

[1] Abraham, S. and Rau, B., "Predicting Load Latencies using Cache Profiling," Tech. Rep. HPL-94-110, Hewlett Packard Laboratories, Dec. 1994.

[2] Abraham, S., Sugumar, R., Rau, B., and Gupta, R., "Predictability of Load/Store Instruction Latencies," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pp. 139–152, Dec. 1993.

[3] Agarwal, A. and Gupta, A., *Temporal, Processor, and Spatial Locality in Multiprocessor Memory References*. Plenum Press, 1989.

[4] Agarwal, A., Horowitz, M., and Hennessy, J., "An Analytical Cache Model," *ACM Transactions on Computer Systems*, vol. 7, pp. 184–215, May 1989.

[5] Allen, R. and Kennedy, K., *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[6] Annavaram, M., Patel, J., and Davidson, E., "Data Prefetching by Dependence Graph Precomputation," in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 52–61, July 2001.

[7] Argawal, H. and Horgan, J., "Dynamic Program Slicing," in *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 246–256, June 1990.

[8] Bass, M. and Christensen, C., "The Future of the Microprocessor Business," *IEEE Spectrum*, vol. 39, pp. 34–39, Apr. 2002.

[9] Burks, A., Goldstine, H., and von Neumann, J., "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument." Papers of John von Neumann, 1987.

[10] Calder, B., Krintz, C., John, S., and Austin, T., "Cache-Conscious Data Placement," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 139–149, Oct. 1998.

[11] Callahan, D., Kennedy, K., and Porterfield, A., "Software Prefetching," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 40–52, Apr. 1991.

[12] Carlisle, M., *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996. http://www.cs.princeton.edu/~mcc/olden.html (Jul. 2006).

[13] Charney, M. and Reeves, A., "Generalized Correlation-Based Hardware Prefetching," Tech. Rep. EE-CEG-95-1, Cornell University, Feb. 1995.

[14] Chilimbi, T., Davidson, B., and Larus, J., "Cache-Conscious Structure Definition," in *Proceedings of the SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp. 13–24, May 1999.

[15] Chilimbi, T., Hill, M., and Larus, J., "Cache-Conscious Structure Layout," in *Proceedings of the SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp. 1–12, May 1999.

[16] Collins, J., Tullsen, D., Wang, D., and Shen, J., "Dynamic Speculative Precomputation," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pp. 306–317, Dec. 2001.

[17] Collins, J., Wang, H., Tullsen, D., Hughes, C., Lee, Y., Lavery, D., and Shen, J., "Long-range Prefetching of Delinquent Loads," in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 14–25, July 2001.

[18] Cragon, H., *Memory Systems and Pipelined Processors*. Jones and Barlett Publishers, 1996.

[19] Denning, P., "The Working Set Model for Program Behavior," *Communications of the ACM*, vol. 11, pp. 323–333, May 1968.

[20] Edler, J. and Hill, M., "Dinero IV Trace-Driven Uniprocessor Cache Simulator." http://www.cs.wisc.edu/~markhill/DineroIV (Jul. 2006).

[21] Fu, J. and Patel, J., "Stride Directed Prefecthing in Scalar Processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 102–110, Dec. 1992.

[22] Hennessy, J. and Patterson, D., *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2003.

[23] Horowitz, M., Martonosi, M., Mowry, T., and Smith, M., "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors," in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 260–270, May 1996.

[24] Intel Itanium Processors. http://www.intel.com/itanium (Jul. 2006).

[25] International Technology Roadmap for Semiconductors. http://www.itrs.net (Jul. 2006).

[26] Joseph, D. and Grunwald, D., "Prefetching using Markov Predictors," *IEEE Transactions on Computers*, vol. 48, pp. 121–133, Feb. 1999.

[27] Kamble, M. and Ghose, K., "Analytical Energy Dissipation Models for Low Power Caches," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 143–148, Aug. 1997.

[28] Kathail, V., Schlansker, M., and Rau, B. R., "HPL-PD Architecture Specification: Version 1.1," Tech. Rep. HPL-9380 (R.1), Hewlett Packard Laboratories, Feb. 2000.

[29] Kernighan, B. and Ritchie, D., *The C Programming Language*. Prentice Hall, 1988.

[30] Kim, D., Liao, S., Wang, P., Cuvillo, J., Tian, X., Zou, X., Wang, H., Yeung, D., Gikar, M., and Shen, J., "Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors," in *Proceedings of the Second International Symposium on Code Generation and Optimization*, pp. 27–38, Mar. 2004.

[31] Kistler, T. and Franz, M., "Automated Data-Member Layout of Heap Objects to Improve Memory-Hierarchy Performance," *ACM Transactions on Programming Languages and Systems*, vol. 22, pp. 490–505, May 2000.

[32] Klaiber, A. and Levy, H., "An Architecture for Software-Controlled Data Prefetching," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 43–53, May 1991.

[33] KleinOsowski, A. and Lilja, D., "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, vol. 1, Jan. 2002.

[34] Korkmaz, P., Puttaswamy, K., and Mooney, V., "Energy Modeling of a Processor Core Using Synopsys, and of the Memory Hierarchy using the Kamble and Ghose Model," Tech. Rep. CREST-TR-02-002, Georgia Institute of Technology, Feb. 2002.

[35] Liao, S., Wang, P., Wang, H., Hoflehner, G., Lavery, D., and Shen, J., "Post-Pass Binary Adaptation for Software-Based Speculative Precomputation," in *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 117–128, June 2002.

[36] Luk, C. and Mowry, T., "Compiler-Based Prefetching for Recursive Data Structures," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 222–233, Oct. 1996.

[37] McKinley, K. and Temam, O., "A Quantitative Analysis of Loop Nest Locality," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 94–104, Oct. 1996.

[38] Mowry, T., Lam, M., and Gupta, A., "Design and Evaluation of a Compiler Algorithm for Prefetching," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62–73, Oct. 1992.

[39] Nethercote, N., *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, Nov. 2004. http://valgrind.org/ (Jul. 2006).

[40] Ozawa, T., Kimura, Y., and Nishizaki, S., "Cache Miss Heuristics and Preloading Techniques for General-Purpose Programs," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 243–248, Nov. 1995.

[41] Panait, V., Sasturkar, A., and Wong, W., "Static Identification of Delinquent Loads," in *Proceedings of the Second International Symposium on Code Generation and Optimization*, pp. 303–314, Mar. 2004.

[42] Panda, P., Dutt, N., and Nicolau, A., "Memory Data Organization for Improved Cache Performance in Embedded Processor Applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 2, pp. 384–409, Oct. 1997.

[43] Petrank, E. and Rawitz, D., "The Hardness of Cache Conscious Data Placement," in *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 101–112, Jan. 2002.

[44] Phalke, V. and Gopinath, B., "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," in *Proceedings of the 1995 ACM SIGMETRICS joint International Conference on Measurement and Modeling of Computer Systems*, pp. 291–300, May 1995.

[45] Rau, B., *Program Behavior and the Performance of Memory Systems*. PhD thesis, Stanford University, 1977.

[46] Rau, B., "Sequential Prefetch Strategies for Instructions and Data," Tech. Rep. 131, Stanford University Digital Systems Laboratory, Jan. 1977.

[47] Sherwood, T., Sair, S., and Calder, B., "Predictor-Directed Stream Buffers," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pp. 42–53, Dec. 2000.

[48] Standard Performance Evaluation Corporation Benchmark Suite. http://www.spec.org (Jul. 2006).

[49] Spirn, J., *Program Behavior: Models and Measurements*. Operating and Programming Systems Series, Elsevier, 1977.

[50] Steensgaard, B., "Points-To Analysis In Almost Linear Time," in *Proceedings of the SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pp. 32–41, Jan. 1996.

[51] Taub, A., *Collected Works of John von Neumann*. The Macmillan Company, 1963.

[52] Thompson, S., Packan, P., and Bohr, M., "MOS Scaling: Transistor Challenges for the 21st Century," Tech. Rep. Q3, Intel Technology Journal, July 1994.

[53] Tip, F., "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, Sept. 1995.

[54] Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. http://www.trimaran.org (Jul. 2006).

[55] Truong, D., Bodin, F., and Seznec, A., "Improving Cache Behavior of Dynamically Allocated Data Structures," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pp. 322–329, Oct. 1998.

[56] Vanderwiel, S. and Lilja, D., "A Compiler-Assisted Data Prefetch Controller," in *Proceedings of the 1999 International Conference on Computer Design*, pp. 372–377, Oct. 1999.

[57] Wilton, S. and Jouppi, N., "An Enhanced Access and Cycle Time Model for On-Chip Caches," Tech. Rep. 93.5, WRL Research Report, July 1994.

[58] Wu, Y., "Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching," in *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 210–221, May 2002.

[59] Zilles, C. and Sohi, G., "Execution-Based Prediction Using Speculative Slices," in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 2–13, July 2001.